NASA Technical Memorandum 100566

The Preliminary SOL Reference Manual

Stephen H. Lucas
Stephen J. Scotti

JANUARY 1989

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

What this Manual Describes ...

The Sizing and Optimization Language, SOL, a high-level, special-purpose language developed for the solution of sizing and optimization problems. This manual is intended as a reference guide for those who wish to write SOL programs on a VAX computer running the VMS system. An overview of SOL appears in NASA Technical Memorandum 100565 entitled, " The Sizing and Optimization Language, SOL - A Computer Language for Design Problems." This manual is a more detailed reference document.

What this Manual Does Not Describe ...

Detailed tutorial information on how to program, advanced system-related information, or detailed reference information on the VAX/VMS command language. Wherever a more extensive understanding of the operating system is required, readers are referred to the appropriate VAX/VMS documentation for more information.

The Structure of This Document ...

The installation guide explains how to install SOL on a VAX/VMS computers. Chapter One describes how to compile, link and run SOL programs. Chapters Two through Eleven provide language reference information and syntax rules. Additional information is found in the appendices. Several of the chapters contain two levels of information: introductory and advanced; the advanced level information is clearly delimited.

A Little about the Sizing and Optimization Language (SOL) ...

SOL is a compiled language, and the SOL compiler produces FORTRAN object code. In other words, the SOL compiler translates your SOL program into an equivalent FORTRAN program. The SOL package consists of three elements: the SOL compiler; the SOL object library, that needs to be on your system for SOL to work properly; and this reference manual. Additional command procedures `SOL` and `LSOL` described in Chapter 1 are also included.

SOL was developed primarily for in-house use, so at present SOL runs only on DEC VAX/VMS systems. To support SOL you need a VAX/VMS computer. In addition, your system needs at least two things. First, your system must have an editor (such as EDT) so that you will be able to type your SOL programs. Any editor will do, but full-screen editors are generally better for composing programs. Second, you will need the VAX FORTRAN compiler. Since the SOL compiler translates SOL into FORTRAN, you will need to compile and link the FORTRAN to execute your program.

The SOL compiler is a Pascal program written using the MYSTRO compiler development system created at the College of William and Mary and available through COSMIC. By utilizing MYSTRO, the SOL compiler was developed fairly rapidly, and includes an error recovery capability

Notation Used in this Manual to Describe Syntax ...

Descriptions of the syntax of SOL statements appear regularly in this manual. The following syntactic conventions are used in these descriptions.

- Words and letters in typewriter type indicate that you should type the word or letter as shown.

- Symbols, parentheses and so forth that appear in the descriptions should be typed as shown.
- Items that appear inside of angle brackets, ⟨ and ⟩, indicate a template to be filled in at your choice. The syntax for each angle bracket template is usually described separately.

Differences between SOL Version 1.22 and previous versions ...

OPTIMIZATION:

The \ is no longer used to prefix Optimizer Options. To make previous SOL programs compatible simply delete all \ symbols from optimizer option selections.

FORTRAN blocks:

`FORTRAN` blocks are allowed in the declaration section of the main program or subroutine.

- ALL `FORTRAN` type declarations **MUST** be placed in `FORTRAN` blocks inside a SOL declaration section. The SOL compiler **does NOT** catch this error, but the `FORTRAN` code created by the compiler COULD FAIL TO COMPILE if this rule is not followed. Simply place all `FORTRAN` type declarations inside the appropriate SOL declaration sections to make previous SOL programs compatible.

COMPONENTs:

A new `ASSEMBLAGE` statement has been added. Please see Chapter 7 for details. Basically, `SUMMARIZATION` variables are no longer declared in the declaration section of a main program or subroutine, and the outermost `COMPONENT` now has a distinct `ASSEMBLAGE` syntax with its OWN declaration section for `SUMMARIZATION` variables.

- The way summarization variables are declared has changed.
- The way iteration variables are declared has changed.

Simply make the outermost `COMPONENT` an `ASSEMBLAGE` statement and move summarization variable declarations into the `ASSEMBLAGE` declaration section to make previous programs compatible. Also if iteration variables are used, they must be declared according to the new syntax in Chapter 7.

DECLARATION sections:

`SUMMARIZATION` variables **CANNOT** be declared inside a main program or a subroutine declaration section. A new `ASSEMBLAGE` statement now declares `SUMMARIZATION` variables.

Table of Contents

Installation
Installing SOL on a VAX/VMS Computer

The Sizing and Optimization Language is installed on a VAX/VMS computer in several steps. First, a system environment is created which includes creating a SOL directory. Second, the SOL system is read from the delivery media (i.e. the SOL tape) into the SOL directory. The instructions in this installation guide are geared towards the use of a VAX TK50 magnetic tape cartridge. Finally, several SYSTEM wide parameters and file protections are set.

Installation requires the assignment of logical names and will require system management privileges (i.e. you must login as SYSTEM). Although this guide attempts to make installation simple and easy, system privileges and some understanding of systems management is required. To install SOL, follow the steps below:

1. LOGIN as system:

You must choose the exact disk drives and directory on which to install SOL. The installation instructions given below assume that the installer has system privileges so that system-wide parameters can be set and system login command procedures altered.

2. MOUNT the SOL tape:

Your SOL tape contains a single **save_set** containing all the files required for SOL, including the compiler, optimizer and runtime library. The **save_set** was created using the VAX BACKUP command.

1) **WRITE-PROTECT** the tape by sliding the write-protect switch on the front of the tape to the left. This will prevent any data on the tape from being accidently destroyed during the installation. Instructions for this step can be found in Chapter 5 of the *MicroVMS User's Primer* or in Chapter 1 of the *MicroVMS User's Manual: Part 1.*

2) **LOAD** the tape by inserting it into the tape drive. Instructions for this step can be found in the appropriate VAX documentation. For example, in Chapter 5 of the *MicroVMS User's Primer* or in Chapter 1 of the *MicroVMS User's Manual: Part 1.*

3) **MOUNT** the tape with the following command:

```
MOUNT /FOREIGN tape_drive:
```

Where `tape_drive` is the name (e.g. `$TAPE1`) of the tape drive on which you mounted the tape. The physical name for the TK50 cartridge tape drive is `MUA0`, or if you have two tape drives, `MUA0` and `MUB0`. The system-defined logical name for the TK50 device is `$TAPE1`.

If you wish to identify the files on the SOL delivery tape before proceeding with the installation, issue the following commands AFTER mounting the tape:

```
BACKUP/REWIND/LIST tape_drive:
```

where `tape_drive` is the name of the drive on which the tape was mounted.

3. Create the SOL System Environment

Create a high-level SOL directory so that SOL can be used by several users with the following command:

```
CREATE/DIRECTORY sys$sysdevice:[SOL]
```

Next, set the file protection of this directory such that all users are able to access the files to be placed in it with the following command:

```
SET PROTECTION=(WORLD:RE) sys$sysdevice:[000000]SOL.dir
```

Define the logical name `SOL$DIR` by issuing the command below:

```
DEFINE/SYSTEM SOL$DIR sys$sysdevice:[SOL]
```

Since the `SOL$DIR` logical name must exist at all times, you should add this definition statement to your system startup procedure, usually found in the `sys$manager` directory under the name `systartup.com`:

1) Use an editor (e.g. EDT) to edit the `systartup.com` command procedure.

2) Add the line:

```
$ DEFINE/SYSTEM SOL$DIR sys$sysdevice:[SOL]
```

to the command procedure so that the `SOL$DIR` logical name will always be defined.

3) EXIT the editor, making sure the changes to `systartup.com` are saved.

A command procedure called `SOL_SYMBOLS.COM` is included with the SOL system and contains the standard symbol definitions that allow the SOL user to invoke the SOL compiler and linker as discussed in the SOL user's manual. The following line should be added to the system-wide login command procedure, typically `SYS$MANAGER:SYLOGIN.COM`:

```
$ @SOL$DIR:SOL_SYMBOLS
```

Be sure to add the command to the system-wide login procedure such that the defined symbols will be available for all users, avoiding sections of the login procedure that are only executed for `SYSTEM` or other privileged users.

4. BACKUP the tape into the SOL directory

Once the SOL directory has been created, the SOL files must be copied from the tape into the directory, SOL$DIR. Since the files are stored as a save_set, the VAX BACKUP command is used. Type the following to copy the SOL files:

```
BACKUP/REWIND tape_drive:SOL.BCK SOL$DIR
```

The required files will be copied into the SOL$DIR directory. With TK50 tapes, this operation can be slow so be patient.

Next, set the file protection so that all users are able to access the individual files within the SOL directory with the following command:

```
SET PROTECTION = (WORLD:RE) SOL$DIR:*.*
```

To verify that the files have been properly read from the tape, issue the following command:

```
DIRECTORY/SIZE SOL$DIR
```

The following should be displayed:

```
Directory SYS$SYSDEVICE:[SOL]

DVBOUNDS.FOR;1              4
DVNORM.FOR;1                3
LINKSOL.COM;1               9
OPTIMIZER.FOR;1          1008
OPTIMIZER.OBJ;1           550
OPT_OUTPUT.FOR;1           42
RUNERR.FOR;1                6
RUNSOL.COM;1                2
SOL_COMPILER.EXE;1        378
SOL_COMPILER.PAS;1       1020
SOL_LIB.OLB;1             891
SOL_SYMBOLS.COM;1           1
UNNORMALIZE.FOR;1           3

Total of 13 files, 3917 blocks.
```

If these files are missing, check your BACKUP command for typing errors.

5. DISMOUNT the SOL tape

To dismount the tape from the tape drive, type the following:

```
DISMOUNT tape_drive
```

where tape_drive is the name of the tape drive on which you mounted the tape (e.g. $TAPE1). Once the tape dismount is completed, the tape can be removed.

6. VERIFICATION

This completes the installation process. To verify that things are working:

1) LOGOUT as SYSTEM.

2) LOGIN as a normal user and CREATE a simple SOL program using an editor. For example:

```
PROGRAM test
  PRINT 'this is a test'
END test
```

Save the program in a file named `test.sol`

3) Compile the program with the command `SOL test`.

4) Link the program with the command `LSOL test`. You will be prompted for external file names; simply type a carriage-return in response to the prompt.

5) Run the program with the command `RUN test`.

If the program compiles, links, and runs (producing the output `this is a test` for the example above), then it is reasonable to assume SOL has been installed properly. One might also try shutting the machine down, and then repeating steps 2) through 5) above to further verify the installation.

7. User Authorization

SOL's macro features are implemented with files. As a result, it is possible that SOL programs will require many files to be opened at once. VAX/VMS systems set a limit on the number of files that a user can have opened at once. When a SOL program exceeds the VAX/VMS limits. a system error message will be displayed and the SOL program will halt. One can avoid this problem by increasing the number of files a user is permitted to have open through the use of the `AUTHORIZE` utility, described in the *MicroVMS User's Manual, Part II, Appendix AUTH* . To increase the file limits, do the following:

1) Login as SYSTEM.

2) Switch to the proper directory and run the AUTHORIZE utility with the following commands:

```
SET DEFAULT SYS$SYSTEM

RUN AUTHORIZE
```

3) The `UAF>` prompt symbol should appear. Increase the file limits for EACH SOL user who needs a higher limit with the following command:

```
MODIFY user /fillm=200
```

where "user" is the login name of the user requiring an increased file limit. The AUTHORIZE utility should return the message:

```
%UAF-I-MDFYMSG, user record(s) updated
```

to signify the necessary changes have been made.

4) Increase the number of bytes allowed in buffered I/O operations with the command:

```
MODIFY user /bytlm=10000
```

where "user" is the login name of the user requiring an increased byte limit. The AUTHORIZE utility should return the message:

```
%UAF-I-MDFYMSG, user record(s) updated
```

5) The changes can be verified with the command:

```
SHOW user
```

where "user" is the login name of the user.

6) Exit the AUTHORIZE utility with the command:

```
EXIT
```

It will not be necessary to increase the "opened file" limits unless macros are used extensively.

Chapter 1
Compiling, Linking, and Executing SOL Programs

SOL is a compiled language. After creating a SOL source program with your favorite editor, you must *compile* the program. The compilation process translates your SOL program into an equivalent FORTRAN program. A computer program, the *compiler*, does the translation. You compile your program with the SOL command procedure described below. This command invokes the SOL compiler to translate your SOL program into FORTRAN. After using the SOL command to compile your program, you must use the LSOL command procedure to *link* your program. The LSOL command compiles the FORTRAN program, and links it with the SOL library routines. The result of the LSOL command is an *executable* version of your program, a version which is ready to be run on the computer. Finally, the VAX RUN command is used to run your program.

In summary, to create and run a SOL program, do the following:

1) - Write a SOL program.
2) - Use the SOL command procedure to compile the SOL program
3) - Use the LSOL command procedure to link the SOL program.
4) - Use the VAX RUN command to run the program.

EXAMPLES:

For instance, to compile, link, and execute a SOL program named `testsol.sol`, type the following sequence of commands:

```
SOL testsol
LSOL testsol
RUN testsol
```

The SOL command will invoke the SOL compiler which translates `testsol` into an equivalent FORTRAN program named `testsol.for`. The LSOL command invokes the FORTRAN compiler to compile `testsol.for`, and links the result with needed library routines.

This chapter describes how to use the SOL and LSOL command procedures. The chapter is divided into four sections:

1.1 - Discusses the SOL command
1.2 - Discusses the LSOL command
1.3 - Discusses the VAX RUN command
1.4 - Discusses an example LISTING produced by the SOL compiler

1.1 COMPILING A SOL PROGRAM: THE SOL COMMAND PROCEDURE

The SOL command invokes the SOL compiler. The primary functions of the SOL compiler are:

1) to translate SOL source statements into an equivalent FORTRAN program and to issue any error messages.

2) to optionally generate a LISTING and/or CROSS-REFERENCE file.

The SOL command is not invoked from the editor, or from your SOL program. Once you have created your SOL program, save the program and exit from the editor. At the system prompt, type SOL and a space followed by the filename of your SOL program, another space and the compiler options. Press the RETURN or ENTER key. This will invoke the SOL compiler.
More formally, the SOL command has the following syntax:

SOL ⟨ SOL file ⟩ ⟨ compiler options ⟩

where:

⟨ SOL file ⟩ is the name of your SOL source program (use standard VAX file names; the suffix `.sol` is assumed).

⟨ compiler options ⟩ are the optional settings for the compiler. These are discussed in detail in Chapter 1, section 1.1.2.

The following restrictions hold for the SOL command:

1) The ⟨ SOL file ⟩ and the ⟨ compiler options ⟩ must appear on the same line.

2) One or more spaces MUST appear between the word SOL and ⟨ SOL file ⟩ and between the ⟨ SOL file ⟩ parameter and the ⟨ compiler options ⟩ parameter.

3) Type a carriage return once you have finished typing the SOL command and parameters.

Thus, the SOL command procedure has two parameters: the name of the SOL source file, and optional qualifiers for the compiler. The following examples illustrate the use of the SOL command procedure:

1) `SOL optimum`

2) `SOL optimum.sol lx`

The name of the program being compiled, the compiler options selected, and the current settings for the compiler options are displayed on the screen after the SOL command is given.

For example, the invocation `SOL Optimum lx` will cause the following to be displayed:

```
*** SOL COMPILER UTILITY ***
******* VERSION 1.00 *******
Compiling OPTIMUM with options LX


SOL Compiler v. 1.2, Current Option Settings are:
  CODE  OPTION         SETTING
   D    PRINT RULES      OFF
   L    LISTING          OFF
   P    PARSE TRACE      OFF
   O    CREATE FORTRAN   ON
   X    XREF             OFF

*** This run of the SOL Compiler Utility is complete ***
```

Further details about the SOL command are found in the following three sections:

1.1.1 – Discusses the ⟨ SOL file ⟩ parameter.

1.1.2 – Discusses the ⟨ compiler options ⟩ parameter.

1.1.3 – Discusses the output of the SOL compiler.

1.1.1 The SOL File Parameter

The ⟨ SOL file ⟩ parameter to the SOL command is the name of the SOL source file containing the program to be compiled. The file must exist in the current directory, or the complete file specification must be given.

The SOL compiler assumes a .SOL suffix if no suffix is explicitly given in the ⟨ SOL file ⟩ parameter. For instance, the SOL command:

```
SOL filename
```

is equivalent to using the SOL compiler with the following command:

```
SOL filename.sol
```

Therefore, it is a good idea to suffix your SOL files with the suffix, ".SOL." This suffix will distinguish SOL files from other programs, and save typing when using the SOL command to compile your SOL programs.

1.1.2 The SOL Compiler Options

Compiler options are specified in the ⟨ compiler options ⟩ parameter. The SOL compiler has default settings, so ⟨ compiler options ⟩ can be left blank.

The ⟨ compiler options ⟩ parameter is one of the following:

1) nothing – use the default settings. The default settings are:

l (listing)	ON
o (create FORTRAN code)	ON
x (cross-reference)	ON
p (parse trace)	OFF
d (print rules)	OFF

2) l – turn off the listing option
3) o – turn off the generation of FORTRAN code
4) x – turn off the cross-reference option
5) p – turn on the parse tracing
6) d – turn on the printing of parse rules
7) one or more of 2) ... 6), not separated by spaces or carriage returns. E.g., `lpd` or `lox` or `lo`

Each of the options 2) through 6) is discussed in detail in the five sections that follow.

1.1.2.1 The Listing Option, L

The SOL compiler produces a source listing file by default. This file has the same name as the source file, but with the suffix ".list". For example, compiling "example.sol" will produce a listing file named "example.list".

The Listing option turns off the SOL compiler's listing option, so that no listing is produced. Since the cross-reference listing goes into the listing file, turning off the compiler's listing option also turns off the cross reference option. For example, the following command turns off the SOL compiler's generation of a listing:

```
SOL example l
```

A sample listing is discussed in Chapter 1, section 1.4.

1.1.2.2 The Cross-reference Option, X

The SOL compiler produces a cross-reference index of all variables in the compiled SOL program by default. The cross-reference index appears in the listing file, also produced automatically. The cross-reference index lists each variable, and the line numbers where the variable is accessed.

Invoking the Cross-reference option turns off the SOL compiler's cross-reference in the listing. Since the cross-reference information is produced in the listing, turning off the listing option will also turn off the cross-reference option.

The three example invocations below turn off the cross-reference option:

1) SOL **example x**

2) SOL **example l**

3) SOL **example lox**

A sample cross-reference index is discussed in Chapter 1, section 1.4.

1.1.2.3 The Create FORTRAN Option, O

As a default, the SOL compiler automatically produces an object file containing a FORTRAN program equivalent to the compiled SOL. The object file contains the FORTRAN output from the SOL compiler. The name of the file containing the FORTRAN output is the same as the SOL source file, except it has a ".for" suffix. For example, "example_1.sol" will produce a FORTRAN output file named "example_1.for"

The Create FORTRAN option turns the SOL compiler's FORTRAN code generator off, so that FORTRAN output will **not** be produced. To turn off the code generation, the O option is specified. For instance, the following command will turn OFF the code generation of the SOL compiler:

```
SOL example_1 o
```

The FORTRAN output file, produced when the Create FORTRAN option is left on, must be linked with the LSOL command, discussed in Chapter 1, section 1.2.

1.1.2.4 The Parse Trace Option, P

The parse trace option is intended for debugging the SOL compiler, and therefore is not described. Unless modifying the SOL compiler source code, the P option is not needed.

1.1.2.5 The Print Rules Option, D

The print rules option is intended for debugging the SOL compiler, and therefore is not described. Unless modifying the SOL compiler source code, the D option is not needed.

1.2 LINKING A SOL PROGRAM: THE LSOL COMMAND PROCEDURE

The LSOL command invokes the SOL linker. The primary functions of the SOL linker are:

1) to invoke the VAX FORTRAN compiler to compile the FORTRAN output produced by the SOL compiler.

2) to link the output of the FORTRAN compiler to a standard library of routines, and also to link with any other user-provided FORTRAN subroutines needed by the SOL program. This step produces an executable version of the SOL program.

To link a SOL source program, invoke the LSOL command from the terminal. After using the SOL command procedure, at the system prompt type LSOL and a space followed by the name of the file to be linked. Remember that the file being linked is the *output* of the SOL compiler, so no suffix or a .FOR suffix should be used. (e.g. "example" or "example.for")

More formally the invocation has the following syntax:

LSOL ⟨ SOL file ⟩

where:

⟨ SOL file ⟩ is the name of the FORTRAN program generated by the SOL compiler. (your SOL program and the equivalent FORTRAN program, produced by the SOL command, automatically have the same names).

The following restrictions hold for the LSOL command:

1) The word LSOL, and ⟨ SOL file ⟩ must appear on a single line.
2) One or more spaces MUST appear between the word LSOL and ⟨ SOL file ⟩
3) Type a carriage return once finished typing the LSOL command procedure and parameters.

Thus, the LSOL command procedure has one parameter, the name of the SOL object file. For example, the following calls will invoke the LSOL Command Procedure:

1) `LSOL example_1`
2) `LSOL optimum.for`
3) `LSOL optimum`

The LSOL command will prompt you for the names of any external FORTRAN routines called by your SOL program. For example, the command LSOL Optimum will cause the following to be displayed:

```
*** SOL LINKER UTILITY ***
******  VERSION 1.00   *****
Linking OPTIMUM to SOL Library Routines

Please answer Y or N
Do you have external FORTRAN (.obj) files to be linked?
```

In response to this prompt, you should type Y if you have external routines to link and N if you do not. If you answer Y, you will be prompted for the external file's name, with the following prompt:

```
You will be prompted for file names.
Enter a single name followed by a carriage return.  You will be prompted for
the next name.  When you have finished, just hit a carriage return at the prompt.

Enter the complete filename ==>
```

Type a filename, and type carriage return. The LSOL command continues to prompt you for additional files until you type a carriage return in response to the `enter filename` prompt. Once this prompting has ended, LSOL will compile the FORTRAN translation of your SOL program and invoke the VAX Linker to create an executable (.exe) image which can be run on the computer. The LSOL command procedure will signal that linking has ended by sending the following message to the screen:

```
*** This run of the SOL Linker Utility is complete ***
```

The following restrictions hold for the external file names:

1) The linker requires ".obj" files produced by the VAX FORTRAN compiler. If a FORTRAN routine is needed by your SOL program, you must compile the FORTRAN to create a ".obj" file. This ".obj" file can now be linked with your SOL program.

2) The linker assumes a ".obj" suffix, so it need not be supplied. For example, to link "example_1.obj" either of the following are valid file names:

   "example_1.obj" or "example_1"

3) The linker defaults to the current device and directory. If a different device or directory is desired, this must be specified in the file name.

The executable image will be created in the current directory. This image will have the same name as the file name passed to the LSOL command procedure, except the ".for" suffix is replaced with a ".exe" suffix. For example, `LSOL optim` and `LSOL optim.for` both produce an executable image named, "optim.exe"

1.3 EXECUTING A SOL PROGRAM: THE RUN COMMAND

Once you have compiled your program using the SOL command and linked it using the LSOL command, the last step is to run your program. Simply type:

`RUN` ⟨ file name ⟩

where:

⟨ file name ⟩ is the name of the ".exe" version of your SOL program, produced by the LSOL command. The ".exe" suffix is assumed, so it need not be supplied. For example, to execute "test.exe" you can type either of the following:

`RUN test` or `RUN test.exe`

This command is the VAX RUN command, which is used to execute most programs on VAX/VMS systems.

1.4 SAMPLE SOL LISTING

A complete SOL compiler listing has three parts:

1) A source code listing, generated by leaving the L option on.

2) A cross-reference listing, generated by leaving the X option on.

3) A list of error messages outlining where errors have been encountered in your SOL program.

The source code listing for the main program appears first, followed by the cross-reference information (if X option selected) for the main program. The main program cross-reference is followed by a series of source listing/cross-reference pairs, one pair for each subroutine. The last thing to appear in the compiler listing is the error messages.

On the following pages, a complete SOL compiler listing appears. After the SOL compiler listing, three discussion sections appear that explain the compiler listing. The italicized numbers that appear in the listing correspond to the numbered explanations in the sections that follow:

1.4.1 – Discusses the source listing

1.4.2 – Discusses the cross-reference information

1.4.3 – Discusses the error messages

```
SIZING AND OPTIMIZATION LANGUAGE 1 (SOL) V1.22 2 SOURCE LISTING 3  21-JUN-1988 4 10:53:47.88 5 PAGE 1 6
 LINE# 7 SOURCE LINE 8

    1   : PROGRAM Opt_Beam
    2   : !
    3   : ! Determine the optimimum Geometry for min weight beam
    4   : !
    5   : DECLARE
    6   :   SUBROUTINE (Rho_max) = calc_rho(max, depth)
    7   : END DECLARE
    8   : ! constants for beam weight calculation.
    9   :   Fs_Allow    = 60000
   10   :   Fb_Allow    = 100000
   11   :   length      = 40          ! beam length in inches
   12   :   load        = 100         ! uniform load in lbs per inch
   13   :   steel       = .3
   14   :   t_min       =.1
   15   :   material_wt = 7
   16   :   cap_width     = 3
   17   :
   18   :   OPTIMIZE Beam_Weight
   19   :   USE
   20   :     depth = 20 In [.25, ]
   21   :     cap_thickness = t_min In [t_min, ]
   22   :     web_thickness = t_min In [t_min,  ]
   23   :
   24   :   ! constraints
   25   :     Rho_b_max .lt. Fb_Allow
   26   :     web_stress .lt. Fs_Allow
   27   :   END USE
   28   :
   29   :   ASSEMBLAGE beam (0, 'Beam')
   30   :   SUMMARIZE
   31   :      area
   32   :   END SUMMARIZE
   33   :
   34   :     COMPONENT top_cap(1, 'Top Cap')
   35   :       area = length * cap_width * cap_thickness
   36   :     END top_cap
   37   :     COMPONENT base_cap (1, 'Base Cap')
   38   :       area = length * cap_width * cap_thickness
   39   :     END base_cap
   40   :     COMPONENT web (1, 'Web ')
   41   :       area = depth * web_thickness * length
   42   :     END web
   43   :   END beam
   44   :
   45   :   M_max       = ((load*length)**2)/8
   46   :   I           = (area@top_cap@beam*(depth**2))/2
   47   :   (Rho_b_max) = calc_rho(M_max, depth, I)
   48   :   web_stress  = load*length/(2*area@web@beam)
   49   :   Beam_Weight = area@beam * steel
   50   :   cap_area    = area@top_cap@beam + area@base_cap@beam
   51   :   web_area    = area@web@beam
   52   :
   53   :  END OPTIMIZE
   54   :   Print 'cap area      ', cap_area      : f 5.2
   55   :   Print 'Web thickness ', web_thickness : f 5.2
   57   :   Print 'web area      ', web_area      : f 5.2
   58   :   Print 'Beam Depth    ', depth         : f5.2
   59   : End Opt_Beam
```

*** CROSS REFERENCE FOR MAIN PROGRAM *** 10

*** ASSEMBLAGE NESTING STRUCTURE *** 11

```
BEAM
     TOP_CAP
     BASE_CAP
     WEB
```

*** SUMMARIZATION VARIABLES AND EXPRESSIONS *** 12

Summarization Variable Name	Line Numbers
AREA	32

*** Optimization Listing *** 13

```
OPTIMIZATION NAME :  BEAM_WEIGHT
DESIGN VARIABLES  :  DEPTH, CAP_THICKNESS, WEB_THICKNESS
CONSTRAINTS       :  RHO_B_MAX, WEB_STRESS
```

Variable name, Component Nesting Appended	Line Numbers 14
BEAM_WEIGHT	49
CAP_AREA	50 54
CAP_THICKNESS	21 35 38
CAP_WIDTH	16 35 38
DEPTH	20 41 46 47 57
FB_ALLOW	10 25
FS_ALLOW	9 26
I	46 47
LENGTH	11 35 38 41 45 48
LOAD	13 43 46
MATERIAL_WT	15
M_MAX	45 47
RHO_B_MAX	47
STEEL	13 49
T_MIN	14 21 22
WEB_AREA	51 56
WEB_STRESS	48
WEB_THICKNESS	21 41 55

Variables for Assemblage BEAM 15

Variable name, Component Nesting Appended	Line Numbers
AREA@ BEAM	49

Variables for Component TOP_CAP

Variable name, Component Nesting Appended	Line Numbers
AREA@ TOP_CAP@ BEAM	35 46 50

Variables for Component BASE_CAP

Variable name, Component Nesting Appended	Line Numbers
AREA@ BASE_CAP@ BEAM	38 50

Variables for Component WEB

Variable name, Component Nesting Appended	Line Numbers
AREA@ WEB@ BEAM	41 48 51

```
SIZING AND OPTIMIZATION LANGUAGE (SOL) V1.22 SOURCE LISTING 16      21-JUN-1988  10:53:47.88        PAGE  3
  LINE#     SOURCE LINE

     60    : SUBROUTINE (Rho_max) = calc_rho(max, depth, I)
     61    :   Rho_max    = (max * depth) / ( 2 * I)
     62    : END calc_rho
     63    :
```

```
SIZING AND OPTIMIZATION LANGUAGE (SOL) V1.22 CROSS REFERENCE 21-JUN-1988 10:53:47.88          PAGE  4


*** CROSS REFERENCE FOR SUBROUTINE *** CALC_RHO 17

Variable name, Component Nesting Appended   Line Numbers

DEPTH                                         60   61
I                                             60   61
MAX                                           60   61
RHO_MAX                                       61
```

```
SIZING AND OPTIMIZATION LANGUAGE (SOL) V1.22 SOURCE LISTING  21-JUN-1988 10:53:47.88          PAGE  5
  LINE#     SOURCE LINE


0     2 ERRORS FOUND. 18
1     0 WARNINGS ISSUED.
```

```
SIZING AND OPTIMIZATION LANGUAGE (SOL) V1.22   ERROR MESSAGES 19   21-JUN-1988  10:53:47.88   PAGE       6
  LINE#     SOURCE LINE

     47    :   (Rho_b_max) = calc_rho(M_max, depth, I)
 *** ERROR   ^ SUBROUTINE ARGUMENT NUMBER NOT MATCH DECLARATION
     60    : SUBROUTINE (Rho_max) = calc_rho(max, depth, I)
 *** ERROR                              ^ SUBROUTINE ARGUMENT NUMBER NOT MATCH DECLARATION
```

1.4.1 THE SOURCE LISTING

This section explains the format of the compiler source listing in detail. The italicized numbers in the explanations below refer to the italicized numbers that appear in the sample listing from the previous pages.

Each page of the listing begins with a title line, consisting of eight elements:

1 A SOL header message.

2 The version number of the SOL compiler.

3 A subtitle that describes which part of the listing is found on this page. In the case of the source listing the words, "SOURCE LISTING," appear.

4 The date the listing was created.

5 The time the listing was created.

6 The page number for the complete listing.

7 A line number column.

8 A source line column.

The compiler listing begins with the main program source listing, as on page one. The lines from your SOL program appear, and are numbered for referencing. The line numbers that appear in the cross-reference and error messages, refer to these line numbers in the source listing.

Following the main program source listing, the cross-reference for the main program (if the cross-reference option was chosen) appears. Next, the source listing for the subroutine implementation appears, *16* on page 3 of the listing. The subroutine source listing is followed by its own cross-reference. If other subroutines were used, a source listing/cross-reference pair for every subsequent subroutine implementation would also appear.

The last page of the source listing will be blank if no errors occur, or will contain information on the number of errors and the number of warnings *18* that appeared followed by a listing of the error messages *19*.

1.4.2 THE CROSS-REFERENCE LISTING

The cross-reference listing follows the source listing. The cross-reference will begin on a new page, and will be indicated by the words, "CROSS REFERENCE," in the page title as in *9*.

The cross-reference listing consists of the following:

10 A title indicating that the cross-reference is for the main program. In the case of subroutines, a different title is used to indicate the name of the subroutine, as in the case of *17*.

11 An **ASSEMBLAGE** section appears, which illustrates components that make up the assemblage. If no components appear, this section is left blank.

12 A list of the **ASSEMBLAGE** summarization variables and summarization expression variables appears, along with the line number where they were declared.

13 An optimization section appears, which lists the objective function, design variables and constraints for each optimize statement in your SOL program.

14 A variable listing appears, which lists the variables and the lines where they were used. (**ASSEMBLAGE** and **COMPONENT** variables are listed separately, see *15* below).

15 A variable listing for the assemblage and associated components appears.

Following the cross-reference of the main program, the source listing for the subroutine appears, followed by its own cross-reference section as on pages 3 – 4 of the listing. Source listing/cross-reference pairs continue to alternate for each subsequent subroutine.

If the cross-reference option is not selected, only a source listing will appear in the listing. Also, if the listing option is not selected, the cross-reference option defaults to "not selected," and no cross-reference will be produced.

1.4.3 The Error Listing

The next to last page of the source listing *18* gives the number of error and warning messages that were issued, and is followed by a list of the messages in an error message section. The error message section is clearly delimited as it begins on a new page, and the words "ERROR MESSAGES" appears in the page header *19*. The line number, line, and message issued for each error follow. The "up-arrow" mark before the issued message points to the general location of the error in the line.

Chapter 2
Introduction to SOL

This chapter is divided into the following sections:

2.1 – Presents an overview of SOL, and illustrates the structure of a SOL program.

2.2 – Discusses SOL's lexical elements – the character set, special symbols, reserved words, identifiers and numbers.

2.3 – Explains how to document a SOL program using comments.

2.4 – Explains the use of SOL's continuation lines.

2.1 AN OVERVIEW OF SOL

SOL is a high-level, special-purpose language that has been developed for use under the VAX/VMS operating system. Four basic features describe SOL:

1) **Conventional features:** SOL has many features of "conventional" languages such as `FORTRAN` or `Pascal`. SOL offers variables; math-operators; built-in mathematical functions; program control statements for loops and if/then/else logical branching statements; subroutines; and some `PRINT` statements to allow the output of values.

2) **Optimization features:** One of SOL's purposes is to make the computer implementation of a numerical optimization problem as simple and error-free as possible. There should be no confusion; SOL is **not** intended as a language for the development of numerical methods of mathematical optimization. Rather, SOL's purpose is to provide a language in which to write code which *applies* existing methods of numerical optimization to *solve* an optimization problem. One writes SOL code to apply optimization. At present, the methods of numerical optimization implemented in the ADS† optimization routine are available for use within SOL programs. In terms of its optimization capability, SOL can be considered as a sophisticated shell around an optimization routine.

† *ADS - A FORTRAN Program for Automated Design Synthesis — Version 1.10,* NASA Contractor Report 177985, Grant NAG1-567, 1985 by G.N. Vanderplaats

3) **Sizing Features:** SOL includes features to facilitate a type of engineering systems modeling herein referred to as *sizing*. In this manual, sizing is defined as the modeling of a system *as the simple sum of its parts* with respect to some special *summarization variables*. For example, an airplane can be modeled and *sized* for weight; we model the major parts of the airplane, its systems and structural components, along with the interaction between the parts. The model is constructed so that *the weight of the entire airplane can be determined by summing the weight of its parts*, the systems and structural weights. Likewise the weight of any part of the airplane can be determined by summing *its* parts. In this case, "weight" is considered a *summarization variable*. SOL aids the modeling of such "assemblages;" by allowing the user to create such models, and automatically computing the necessary summations.

4) **FORTRAN Interface:** SOL provides a `FORTRAN block` feature, which allows one to write **FORTRAN** code within a SOL program, or to interface with an existing **FORTRAN** code.

- Conventional SOL features combine with its specialized optimization and sizing capabilities to create programs.
- SOL's compiler also provides an important error-checking capability. The compiler not only checks for syntax errors, but also uses knowledge about sizing and optimization to give specialized error messages.
- As a convention in this manual, a description of the correct use of a SOL feature is followed by a list of restrictions on usage which defines the error-checking of the compiler.

The following sections introduce the main elements of SOL:

2.1.1 Data Types

Every SOL variable has a data type. A data type classifies a variable, determining both the range of values the variable can have, and the operations which can be performed on it.

SOL provides only three kinds of predefined types:

- `REAL` (8 Bytes long)
- `INTEGER` (4 Bytes long)
- `LOGICAL`

- SOL does NOT allow user-defined types.
- SOL does NOT provide structured types such as arrays or records.

- Chapter 3 describes SOL data types in greater detail.

SOL identifiers are given a data type in two ways:

1) Explicit: Explicit type declarations can appear in the declaration sections of a SOL program, or in a subroutine formal parameter list. Explicit declarations give variables a type.

2) Implicit: If a SOL identifier is not explicitly declared, the variable is implicitly declared as type **REAL** when it is initialized.

Chapter 5 offers an in depth discussion of type declarations.

2.1.2 VARIABLE INITIALIZATION

SOL distinguishes between initializing a variable and declaring a variable.

- A variable is declared when it is associated with a data type.
- A variable is initialized when it first receives a value.
- SOL requires every variable to be initialized before you access its value.

The most common means of accessing a variable's value is using the variable on the right-hand side of an assignment statement. Variables can be initialized in six ways:

1) most variables are initialized by appearing on the left-hand side of an assignment statement. (See Chapter 6, section 6.1)

2) subroutine independent parameters are initialized for use within the subroutine when the subroutine is called and passed values. (See Chapter 9, sections 9.2-9.3)

3) optimization design variables are initialized when they are bounded (given a range of possible values) in the **USE** section of an optimize statement. (See Chapter 8)

4) summarization variables of composite **COMPONENT**s are implicitly initialized at the end of a **COMPONENT**. (See Chapter 7)

5) Iteration variables are initialized in the **ITERATE** section of a **COMPONENT** statement (See Chapter 7, section 7.1.3)

6) variables are initialized when they are returned as the dependent variables from a subroutine call. (See Chapter 6, section 6.7, or Chapter 9)

Variable initialization takes place in the statement section of a SOL program the first time a variable gets a value. Detailed discussions of each variable initialization method can be found in the sections named above.

The *block* in which a variable is initialized determines the variable's *scope*, where the variable can be accessed. Each of the following is a block in SOL:

1) The main program

2) Subroutines

3) IF/THEN/ELSE statements

4) ASSEMBLAGEs and related COMPONENTs

Block: The chief characteristic of a block is that a variable can be initialized inside a block, and remain uninitialized outside it. Some blocks can be nested inside each other.

Scope: The scope of a variable determines where the variable can be accessed, and the rules which decide the scope of the variable are called "scope rules." Scope rules follow from SOL's stringent error checking. The scope rules are designed to insure that variables are always initialized before they are used. For instance, consider the IF/THEN/ELSE statement. One cannot be certain that the statements in the THEN or the ELSE portion of an IF statement will be executed. Statements in the THEN part are executed only when the condition is true, and statements in the ELSE part are executed only when the condition is false. Therefore, it is uncertain whether a variable initialization which only appears in either the THEN or ELSE portion of an IF statement, will occur. SOL will not allow a variable to be used unless its initialization is certain. Hence, the scope rules require that only a local variable be initialized when it is not certain that the initialization will actually take place.

- In general, the scope of a variable includes the block where the variable was initialized, and any blocks nested inside the initializing block. Outside of this scope, the variable is uninitialized and cannot be accessed.

Chapter 11 offers a detailed discussion of scope rules, and the scope rules for each block are discussed when the block is described.

2.1.3 Executable Statements

Statements include conditional branching, loops, assignments, optimizations, ASSEMBLAGE descriptions, print statements, subroutine calls, and FORTRAN blocks. The statement section ends when the main program or subroutine ends. The following table gives a representitive list of the SOL statements and their use:

Table 2-1

Calculation Statements	*Description*
Assignment	Assigns a value to a variable.
Expression	Combines variables, operators, and/or functions to give values. Expressions can only appear as part of other statements.
Control Statements	
Conditional DO loop	Repeats statements while a condition is TRUE.
Iterative DO loop	Repeats statements a specified number of times.
IF/THEN/ELSE	Branches based on a logical decision.
Subroutine call	Calls a SOL subroutine.
Declaration Statements	
Subroutine declaration	Declares a subroutine and its parameters.
Variable declaration	Declares a variables type.
Description Statements	
ASSEMBLAGE or COMPONENT	Describes a sizing model.
OPTIMIZE	Describes an optimization problem.
Miscellaneous Statements	
FORTRAN block	Incorporates FORTRAN code into a SOL program.
?INCLUDE	Include a file into the SOL program.
Macro call	Use a macro abbreviation.
Macro definition	Define a macros abbreviation.
Output Statements	
PRINT	Output a value or optimization result
SUMMARIZE	Output individual component values for a sizing model

SOL statements are fully described in Chapter 6.

2.1.4 SUBROUTINES

SOL allows you to group declarations and executable statements into subroutines. Subroutines are a convenient way to organize a program, because you can isolate individual tasks a program must accomplish by coding each task as a subroutine.

- Subroutines must be declared in the declaration section (Chapter 5) of the main program, before they are used. The declaration consists of the subroutine name, and the number and types of parameters.
- Subroutine implementations, in which the actual code for the subroutine is given, appear after the main program's body.

- SOL subroutines can call each other, but *recursion* is not allowed; SOL subroutines cannot call themselves directly or indirectly.
- Subroutines can declare local variables, but not local subroutines.
- SOL has no global variables; SOL subroutines cannot access variables initialized in the main program, except through explicit parameter-passing.
- Chapter 9 describes subroutines more fully.
- Chapter 6, section 6.7 provides details on subroutine calls.
- Chapter 5 describes subroutine declaration.

2.1.5 Structure of a SOL Program

A SOL program has the following visual structure:

```
Program Header
   Optional Declaration Section
   Statement Section
End Program Footer
   Optional Subroutine Implementation Section
```

The declaration and subroutine implementation sections are optional, and can be left out of your SOL programs under certain conditions. All the other sections must appear.

The following sample SOL program illustrates the structure of SOL programs in general. In the example, the structures outlined above have been noted by SOL comments, which follow the symbol, !.

```
PROGRAM example              ! this is an example program header.  Special
                            ! SOL words are shown in uppercase letters

DECLARE                     ! The declaration section, appears in this
  INTEGER a_var             ! program. A variable is declared to be
  SUBROUTINE (y) = test(x)  ! of type INTEGER and a subroutine is declared
END DECLARE

a_var = 4                   ! The statement section, and here is
(a_var) = test (a_var)      ! a subroutine call

END example                 ! The end program footer

SUBROUTINE (y) = test (x)   ! the start of the subroutine
                            ! implementation section, only one
  IF x .eq. 4 THEN          ! subroutine is implemented in this
    print 'x equals 4'      ! sample program
  ENDIF
  y = x + 1
END test                    ! end of subroutine implementation
```

Thus, to write a SOL program:

1) Give the program header, consisting of the word **PROGRAM**, followed by the name of your program.

2) Write a declaration section, delimited by the reserved words **DECLARE** and **END DECLARE**, if desired.

3) Supply the SOL statements that make up the statement section. Indicate the end of the statement section with the end program footer, which consists of the word **END**, followed by your program name.

4) If subroutines have been declared, supply their implementations in the subroutine implementation section.

Blank lines and comment lines, lines that consist of nothing or a comment respectively, can appear almost anywhere in a SOL program including before the program header and after the end of the main program or subroutine implementation.

In the chapters of this manual, specific details are given about the declaration section, SOL statements and statement section, subroutines, and the relationship between these parts. But, all SOL programs, no matter how complicated, will still have the basic structure outlined above.

2.2 LEXICAL ELEMENTS

A SOL program is composed of lexical elements. Lexical elements consist of a single character (individual symbols like parenthesis or mathematical operators), or a collection of characters (words that have a special meaning in SOL). Each character must be a member of SOL's character set, described in section 2.2.1.

- Some characters act as special symbols in SOL, representing statement delimiters, operators, or elements of the language syntax. These special symbols are presented in section 2.2.2.
- Some words in SOL are reserved for the names of SOL language constructs, statements, and operations. The SOL reserved words are listed in section 2.2.3.
- Some words in SOL are identifiers that are created by the user to name variables, subroutines and so forth. Section 2.2.4 explains how identifiers are formed.

2.2.1 CHARACTER SET

SOL uses the extended ASCII character set used by VAX Pascal. The SOL compiler assumes that the horizontal tab will be represented by the ASCII character numbered by decimal number 9.

The SOL compiler does not distinguish between uppercase and lowercase; for example the word OPTIMIZE has the same meaning when written in any of the following ways:

OPTIMIZE

optimize

OpTImiZe

2.2.2 Special Symbols

SOL utilizes a number of special symbols which are listed in the following table, along with a short English description of their meaning in SOL. For symbols which are composed of more than a single character, the characters must be contiguous and cannot be separated by spaces.

Table 2-2: Special Symbols

Symbol	*Symbol Description*
!	comment delimiter symbol
%	percentage symbol
(	open parenthesis, often delimits arithmetic expressions
)	close parenthesis, often delimits arithmetic expressions
*	multiplication
**	exponentiation
+	addition
,	comma
-	subtraction
/	division
/*	beginning delimiter of FORTRAN blocks in SOL programs
'	quote symbol for SOL strings
*	ending delimiter of FORTRAN blocks in SOL programs
:	colon
=	assignment symbol
&	continuation symbol for overlong lines
?	macro delimiter, indicates where a macro begins
{	macro replacement text opening delimiter
}	macro replacement text closing delimiter
[	optimization design variable bounds opening delimiter
]	optimization design variable bounds closing delimiter
.NOT.	logical operator, boolean negation
.AND.	logical operator, boolean conjunction
.OR.	logical operator, boolean disjunction
.TRUE.	boolean true
.FALSE.	boolean false
.LT.	relational operator "less than"
.LE.	relational operator "less than or equal to"
.EQ.	relational operator "equals"
.NE.	relational operator "not equal to"
.GE.	relational operator "greater than or equal to"
.GT.	relational operator "greater than"

2.2.3 Reserved Words

In SOL, certain words form the basic SOL language, and are predefined with special meanings that cannot be changed. These words are reserved for the names of statements, data types, and operators. Reserved words can appear in uppercase or lowercase, but the following table shows SOL reserved words in all uppercase letters.

Table 2-2: Standard SOL Reserved Words

ABS	DFP	IF	NONE	SIN
ACTIVE	DIRECTIONS	IN	NORMALIZE	SQRT
ASSEMBLAGE	DO	INITIALLY	NOTHING	STEP
AT	ELSE	INSCRIBED	OBJECTIVE	STRATEGY
ATAN	END	INT	OPTIMIZE	SUBROUTINE
BFGS	ENDDO	INTEGER	OPTIMIZER	SUMMARIZE
BOUNDS	ENDIF	INTERPOLATION	OPTIONS	TAB
COMP	EVERY	INTERPOLATION/EXTRAPOLATION	PENALTY	TAN
COMPONENT	EVERYTHING	ITERATE	PRINT	TERMINATION
CONSTRAINTS	EXP	ITERATION	PROGRAM	THEN
CONVEX	EXTERIOR	LAGRANGE	QUADRATIC	USE
COS	FEASIBLE	LINEAR	REAL	VARIABLES
CRITERIA	FIND	LOG	REEVES	VIOLATED
CUBIC	FLETCHER	LOGICAL	SEARCH	WHEN
DECLARE	GOLDEN	MODIFIED	SECTION	
DESIGN	HYPERSPHERES	MULTIPLIER	SEQUENTIAL	

Reserved words can only be used in the contexts for which they are defined. Reserved words cannot be redefined for use as identifiers. (see section 2.2.4 for description of identifiers)

2.2.4 Identifiers

SOL identifiers are used to name variables, ASSEMBLAGEs or COMPONENTs, a SOL program, SOL macros, or a SOL subroutine. However, there is a standard form with which all identifiers must conform to be considered legal:

1) All SOL identifiers must begin with a letter

2) After the starting letter, the rest of the identifier can be a combination of the following elements:

   - letters
   - digits(0...9)
   - underscores(_).

3) SOL is not letter case sensitive, so the letters can be uppercase, lowercase, or some combination of both.

4) Reserved words cannot be used as identifiers.

5) Identifiers MUST be strictly less than 28 characters in length.

Some illustrations of legal identifiers and illegal identifiers follow:

LEGAL – birthday, OuT_tO_LUncH, a234_5k6, First_try, tHe_3rd_value

ILLEGAL – 2_late, 12_hours, a$$time, program, a.answer, ?wrongo

* Note: program is illegal because it is a reserved word.

ADVANCED MATERIAL:

SOL also has extended identifiers. These are only used to access variables initialized inside of an **ASSEMBLAGE**. Extended identifiers consist of a list of identifiers, separated by the @ symbol. Several examples of extended identifiers follow:

weight@shaft a_large_test@my_will golf@noon@can_be@fun

- No blanks can appear between the symbol, @, and the adjacent identifiers.
- Extended identifiers MUST be less than 121 characters in total length.
- See Chapter 7 for more information on ASSEMBLAGEs and extended identifiers.

2.2.5 NUMBERS

SOL numbers can be positive or negative numbers that may include decimal points or be expressed as powers of 10 through the use of scientific notation. In this notation, the letter "E" stands for "times 10 to the power of." Some examples of SOL's representation of integers, reals and scientific notation are given follow:

integer = 12, 0, +2, -4, 1000000, 66, 29

real = 12, 0, 0.0, +0.123, -3.456, .72

scientific notation = 12e2 (equals "12 times 10 to the second" or 1200)
+2e-3 (equals "2 times 10 to the -3rd" or .002)
1.0E06 (equals "1 times 10 to the sixth" or 1000000)

- INTEGER values range from -2,147,483,648 through 2,147,483,648.
- REAL values range from 1.7e38 through 0.29e-38.
- Leading zeros are allowed but not required.
- Commas are not allowed. Thus, one hundred thousand is written as 100000, not 100,000. Unless otherwise specified, SOL will output numbers using scientific notation.

2.3 COMMENTS

It is a good idea to comment your SOL programs, and SOL allows one to use comments freely. You can type anything you want in a comment. The SOL compiler ignores comments completely, so that even reserved words can appear in a comment. The SOL compiler will not do error checking in your comments. There are only two rules for comments:

1) Comments start with an exclamation point, e.g "!"

2) Comments end at the end of the line.

So, once you start a comment on a line, the rest of the line is treated as a comment. Here are some examples of comments:

```
!     this is a comment
!     I can type anything, even 12klf[8_&*)Y)#()J{(MKM{)(#*_)JFM"
!     !!!!!!!!!!!!!!!!!! a comment tooooooooooooo
```

In a SOL program, comments should either appear alone on a line, or after a SOL statement. You cannot put a comment before a SOL statement on a single line, because the entire line will be treated as a comment, and the SOL statement will be ignored just like any other text inside a comment. Some examples follow:

```
a=6 ! comments are effective after statements
! and here is a comment alone on a line
! calculate the area area=width * height
```

The last example above is fine for a comment, but if the text "area=width * height" was meant to be a SOL assignment statement, then there is an error, because it will be ignored as part of a comment.

As a final caution, some keyboards have a double-bar symbol, that looks a little bit like an exclamation point. You cannot use the double-bar symbol as a comment symbol. If you do, your SOL program will be incorrect.

2.4 CONTINUATION LINES

SOL is not entirely free-format, as carriage returns are used as statement separators. There are times when you will not have enough space on your terminal's screen to conveniently finish typing a SOL statement. You would like to continue typing on the next line, but in your case SOL does not allow a carriage return to appear in the middle of the statement. To accomodate this situation, SOL offers a continuation symbol, &.

The continuation symbol, &, should be read as "continued from the previous line." If you place the symbol in column one of a line of text, SOL will treat the text as if you had typed it on the previous line. An example follows:

Legal

```
a = (4 * 3) /
& ((c ** 4) + 2)
```

illegal

```
a = (4 * 3) /
((c ** 4) + 2)
```

There are a few general rules to remember when using the continuation symbol:

1) The continuation symbol, &, MUST be placed in column 1 !! If it is not in column 1, an error will result.

2) There are cases when SOL requires a carriage return after a line. If a continuation symbol appears in column one of the next line, an error will result.

3) You can string together as many lines as desired, using the continuation symbol, &.

4) SOL *strings*, such as `'this is a string'` CANNOT be broken over two lines with the continuation symbol.

5) SOL lines can be at most 120 characters long.

Information about the use of the & symbol in a specific case is sprinkled throughout the manual.

Chapter 3
Data Types

Every SOL variable has a data type. A data type classifies a variable, determining both the range of values the variable can have and the operations which can be performed on it. Furthermore, variables can be combined with operations (e.g. `a + b/2 - c`) to form *expressions.* SOL expressions calculate a value with a certain data type. This chapter's focus is the data types; the discussion of SOL expressions and their types is left to Chapter 4.

- SOL supplies three predefined types: `INTEGER`, `LOGICAL`, and `REAL`.
- No other types, such as arrays, records or user-defined types are available.
- An identifier is declared to be of a certain type in the declaration section (see Chapter 5) of the main program or a subroutine.
- If no explicit declaration is made, identifiers default to be type `REAL`.

This chapter describes the range of possible values and the legal operations for the three predefined types, `INTEGER`, `LOGICAL`, and `REAL`. In addition, the SOL compiler provides some type checking capability which is also discussed. For example, it is an error to assign an `INTEGER` value to a `LOGICAL` variable.

This chapter is divided into four sections:

3.1 – Discusses the `INTEGER` type.

3.2 – Discusses the `LOGICAL` type.

3.3 – Discusses the `REAL` type.

3.4 – Discusses type checking.

3.1 THE INTEGER TYPE

The `INTEGER` type allows positive and negative integer values. A variable must be declared to be of type `INTEGER` in the declaration section of the main program or subroutine (See Chapter 5).

- `INTEGER` values can range from -2,147,483,648 through 2,147,483,647 inclusive.
- An `INTEGER` consists of a series of contiguous decimal digits; no commas or decimal points are allowed.
- Negative `INTEGER` numbers are expressed by placing a minus symbol (−) in front of the number.

- The use of negative **INTEGERs** in complicated expressions may not give the results you expect, see Chapter 4, section 4.3.1 for details.
- The legal operations on **INTEGERs** follow:

```
+, -, *, /, **, .eq., .gt., .ge., .lt., .le., .ne.
```

The following are valid integers in SOL:

```
1200, -1, 0, 2, 24
```

3.2 THE LOGICAL TYPE

The **LOGICAL** type, also known as the boolean type, represents the logical conditions of true and false. Variables are declared to be of type **LOGICAL** in the declaration section of the main program or subroutine (See Chapter 5).

- The symbol, **.true.**, represents a logically true condition.
- The symbol, **.false.**, represents a logically false condition.
- A **LOGICAL** variable can have only one of the two values, **.true.** or **.false.**
- Legal operations on logical values follow:

```
.and., .or., .not.
```

- Relational operators, such as **.GT.** (>), produce logical results. Chapter 4, section 4.2 discusses logical and relational operators.

3.3 THE REAL TYPE

The **REAL** type denotes positive or negative real values. Variables are declared to be of type **REAL** in the declaration section of the main program or subroutine (See Chapter 5).

- All **REAL** values are double-precision, (eight bytes long).
- **REAL** values are allowed to range from ±1.7e38 through ± 0.29e-38 inclusive.
- Variables are assumed to be **REAL** unless declared otherwise.
- Negative **REAL** numbers are expressed by placing a minus symbol (−) in front of the number.
- The use of negative numbers in complicated expressions sometimes will not produce the results you expect, See Chapter 4, section 4.3.1.
- Legal operations on **REAL** variables follow:

```
+, -, *, /, **, .eq., .gt., .ge., .lt., .le., .ne.
```

REAL numbers can be expressed with either of the following notations:

1) **Decimal notation:** Use the set of decimal digits and an optional decimal point. Leading zeros are ignored.

2) **Scientific notation:** Some numbers cannot be conveniently represented with decimal notation. The parts of a REAL number written with this notation are:

   — a REAL number or INTEGER,

   — an upper-case or lower-case "e,"

   — an INTEGER exponent.

The letter "e" stands for "times ten to the power of."

The following are valid REAL numbers in scientific notation, representing the number 237:

Table 3-1:

Scientific Notation	*English Description of Meaning*
237e0	237 times 10 to the 0 power.
2.37e2	2.37 times 10 to the second power.
0.000237e+6	.000237 times 10 to the sixth power.
2370e-1	2370 times 10 to the negative first power.
.237e3	.237 times 10 to the third power.

The following are valid REAL numbers in decimal notation:

6, 6.0, 500, 006, 56.8, .89, 0.5312

3.4 TYPE CHECKING

SOL offers two kinds of type checking:

1) **Assignment Compatibility rules:** Assignment compatibility rules determine the types of data allowed when giving values to a variable. For instance, can a LOGICAL variable be given an INTEGER value?

2) **Operator Compatibility rules:** Operator compatibility rules determine what operations are allowed on variables of a certain type. For instance, can two LOGICAL variables be multiplied together?

This section is divided into two sections:

3.4.1 - Discusses assignment compatibility rules

3.4.2 - Discusses operator compatibility rules

3.4.1 ASSIGNMENT COMPATIBILITY RULES

Assignment compatibility rules apply when giving values to variables, either by assignment or by a subroutine parameter pass. The rules restrict the types of data allowed, as detailed in the following sections:

3.4.1.1 Compatibility rules for regular assignments.

3.4.1.2 Compatibility rules for subroutine parameter passing during subroutine calls.

3.4.1.1 Compatibility Rules for Regular Assignment

Compatibility rules for assignment apply when variables are given a value:

- by assignment statement (Chapter 6, section 6.1);
- when used as optimization design variables (Chapter 8, section 8.1);
- or as iteration variables of an **ASSEMBLAGE** or **COMPONENT** (Chapter 7, section 7.3).

The following table shows the rules of assignment compatibility. The left column gives the type of the variable and the right column gives the types which are assignment compatible.

Table 3-2: Assignment Compatibility Rules

Type of Variable	*Assignment Compatible Type(s)*
INTEGER	INTEGER, REAL
REAL	INTEGER, REAL
LOGICAL	LOGICAL

- Assignments between incompatible types are not allowed and result in a compile-time error message.
- An **INTEGER** variable is assigned the *truncated* **REAL**; for instance, if an **INTEGER** variable is assigned the **REAL** number, 3.99999, the integer variable will receive the value, 3.

3.4.1.2 Subroutine Parameter Passing Assignment Compatibility Rules

The following table shows assignment compatibility rules for the parameter passing during subroutine calls. The left side of the table gives the type of parameter, and the right column lists the types which are assignment compatible.

Table 3-3: Assignment Compatibility for Subroutine Calls

Type of Parameter	*Assignment Compatible Type(s)*
INTEGER	INTEGER
REAL	REAL
LOGICAL	LOGICAL

- Assignments between incompatible types are not allowed.
- Compatibility rules for parameter passing are strict. The variable passed must have the same type as the parameter.
- Further details on subroutine parameters are found in Chapter 5, section 5.2, Chapter 6, section 6.7, and in Chapter 9, section 9.2.

3.4.2 Operator Compatibility Rules

Operator compatibility rules define which operations are allowed on variables of each type. The legal operators for each data type have already been given, and the results are summarized in the following table. The left side of the table lists the type, and the right column displays the legal operations for that type.

Table 3-4:

Type	*Legal Operations*
INTEGER	+, −, *, /, **, .eq., .gt., .ge., .lt., .le., .ne.
REAL	+, −, *, /, **, .eq., .gt., .ge., .lt., .le., .ne.
LOGICAL	.and., .or., .not.

- Note that **INTEGER** and **REAL** types share the same operations. As discussed in Chapter 4, **REAL**s and **INTEGER**s can be mixed in *expressions*, with the resulting value always being of type **REAL**.
- Chapter 4 provides a more detailed discusion of this topic.
- Chapter 10 lists predefined functions and their assignment compatibility rules.

Chapter 4
Expressions

A SOL expression consists of variables or constants combined with operators (e.g. a + b/2 + 32); the expression calculates a value of a certain data type. This chapter's focus is on expressions; a discussion on data types appears in Chapter 3.

SOL has two kinds of expressions:

1) Arithmetic expressions which calculate `INTEGER` or `REAL` values.

2) `LOGICAL` (boolean) expressions which calculate `LOGICAL` values.

A SOL expression is one of the following:

1) a single variable

2) a single constant

3) a single predefined function call

4) a collection of variables and/or constants and/or predeclared function calls, all combined with operators.

- The operators used to form SOL expressions are the arithmetic, relational, and `LOGICAL` operators, all of which are explained in the sections that follow.
- The data type of the expression is determined by the data types of the operators or operands as described in the sections that follow.
- The predeclared functions (See chapter 10 for more details on predeclared functions) are:
  - `ABS, ATAN, COS, EXP, INT, LOG, SIN, SQRT, TAN`
- Expressions cannot be parameters to subroutines.
- Expressions extending over more than a single line in SOL must use the continuation symbol, &, See Chapter 2, section 2.4 for details.

This chapter is divided into two sections:

4.1 - Discusses arithmetic expressions, describing their syntax and action.

4.2 - Discusses `LOGICAL` expressions, describing their syntax and action.

4.3 - Discusses the precedence rules that determine the order in which operators are evaluated.

4.1 ARITHMETIC EXPRESSIONS

Arithmetic expressions calculate **REAL** or **INTEGER** values.

- To form an arithmetic expression, combine numeric data (constants, **INTEGER**, or **REAL** variables) with one or more arithmetic operators
- The use of parentheses in expressions is outlined in section 4.3.3 of this chapter.

The following table lists the arithmetic operators (Also see Chapter 3, section 3.4.2):

Table 4-1:

Operator	*Example*	*Result*
+	a + b	The sum of a and b.
+	+a	The positive value of a.
-	a - b	Subtract b from a.
-	-a	The negative value of a.
*	a * b	The product of a and b.
**	a ** b	a raised to the power of b.
/	a / b	a divided by b.

Although the previous examples show no more than two operands, there is no limit. The following details are also important:

1) Negative exponents must be enclosed in parentheses. E.g. `a**(-b)`

2) Arithmetic operations cannot be applied to **LOGICAL** values.

3) **INTEGER** division is truncated, not rounded. E.g Consider two **INTEGER** variables, c and d. If c=9 and d=8, then d/c = 0, not 1.

4) The value calculated by an arithmetic expression has a data type, which is determined by the types of the operands and operators:
   - If all operands are of type **INTEGER**, and no division operators appear, the resulting value will be of type **INTEGER**. **note:** the predeclared functions can also be operands; predeclared functions return either **INTEGER** or **REAL** values (See Chapter 10).
   - If any operands are **REAL** or if a division operator appears, the resulting value will be of type **REAL**.

5) Operator precedence rules are given in Chapter 4, section 4.3

4.2 LOGICAL EXPRESSIONS

LOGICAL expressions calculate **LOGICAL** values.

- To form a **LOGICAL** expression, combine **LOGICAL** data terms (e.g. **LOGICAL** constants or variables) with one or more **LOGICAL** operators.

- Relational operators test the relationship between two arithmetic expressions and return a LOGICAL value as a result. If the relationship holds, the value .TRUE. is returned, otherwise the value .FALSE. is returned. Thus, relational operators can be used to form LOGICAL data terms which can appear in a LOGICAL expression.

The following table lists the relational operators.

Table 4–2:

Operator	*Example*	*Result*
.eq.	a .eq. b	.true. IF a is equal to b
.gt.	a .gt. b	.true. IF a is greater than b
.ge.	a .ge. b	.true. IF a is greater than or equal to b
.le.	a .le. b	.true. IF a is less than or equal to b
.lt.	a .lt. b	.true. IF a is less than b
.ne.	a .ne. b	.true. IF a is not equal to b

- Relational operators are indivisible units, no spaces or miscellaneous characters can appear between the periods and the letters.
- LOGICAL values cannot be used with relational operators (e.g. .true .ge. c).
- REAL and INTEGER type values can be compared.
- Chapter 4, section 4.3 contains information on operator precedence.

The LOGICAL operators used to combine LOGICAL data terms formed with relational operators and/or LOGICAL constants and/or LOGICAL variables are listed in the following table:

Table 4–3:

Operator	*Example*	*Result*
.and.	a .and. b	.true. IF both a and b are .true. .false. IF either a or b are .false.
.or.	a .or b	.true. IF a or b is .true. .false. IF both a and b are .false.
.not.	.not. a	.true. IF a is .false. .false. IF a is .true.

- LOGICAL operators can only be used with LOGICAL values as arguments.
- Chapter 4, section 4.3 below provides information on operator precedence.

4.3 OPERATOR PRECEDENCE

Precedence rules determine the order in which the operations will be evaluated in expressions. There are three types of precedence rules:

4.3.1 PRECEDENCE RULES FOR ARITHMETIC EXPRESSIONS

The following table lists the precedence order for all operators which can appear in an arithmetic expression:

Table 4–4:

Operator	*Precedence level*
**	first
*, /	second
+ and -	third

- Operators are evaluated in order of precedence listed above.
- In the case of two operators which have equal precedence, evaluation takes place from left to right.
- In the case of the operator, "**," evaluation takes place from right to left. For example:

```
a ** b ** c
```

  is evaluated as

```
a ** ( b ** c ).
```

- Two operators cannot be placed in succession. E.g. `a * -b` is illegal while `a * ( - b )` is legal.
- See Chapter 4, section 4.3.3 discusses the use of parentheses to force precedence.

4.3.2 PRECEDENCE RULES FOR LOGICAL EXPRESSIONS

The following table lists the precedence assigned for all operators that can appear in a `LOGICAL` expression:

Table 4–5:

Operator	*Precedence*
Relational Operators	first
`.not.`	second
`.and.`	third
`.or.`	fourth

- Arithmetic expressions, being compared with relational operators, are evaluated using arithmetic precedence rules before relational operator precedence rules are applied.
- Some `LOGICAL` expressions are evaluated before all subexpressions are evaluated. For example, if a is `.false.`, then `a .and. ( c / b .lt. 12 )` can be determined by testing a, without evaluating `( c / b .lt. 12 )`. This is useful for avoiding division by zero and other problems.
- Two `LOGICAL` operators cannot appear consecutively, unless the second operator is `.not.`

4.3.3 Using Parentheses to force Precedence

Parentheses can appear in arithemetic, relational and `LOGICAL` expressions to alter the normal sequence of evaluation.

- Whatever appears in parentheses is given higher precedence, and evaluated prior to any other operators. Consider these examples:

```
2 ** 3 ** 2 = 2 ** 9 = 512      versus    (2 ** 3) ** 2 = 8 ** 2 = 64

4 + 4/2 = 4 + 2 = 6             versus    (4 + 4)/2 = 8/2 = 4

If a = .false., b = c = .true.
a .and. b .or c = .true.        versus    a .and. (b .or. c) = .false.
```

- The use of parentheses is mandatory when negative or positive operators are used in arithmetic expressions. E.g. `a + -b` is illegal while `a + (-b)` is legal.

Chapter 5
Declaration Section

The declaration section appears immediately after the main program or subroutine header (See Chapter 2, section 2.1.5), and has two functions:

1) Allows the explicit declaration of variables' types.

2) Allows subroutines to be declared

- The declaration section is *optional*; it need not appear.

The declaration section (when it appears) has the following syntax:

```
DECLARE
    ⟨ Variable or Subroutine Decls ⟩
END DECLARE
```

where:

⟨ Variable or Subroutine Decls ⟩ consists of:

- nothing, it is optional
- a variable type declaration (See Chapter 5, section 5.1).
- a subroutine declaration (See Chapter 5, section 5.2).
- a series of variable type and/or subroutine declarations separated by carriage returns.
- The ⟨ Variable or Subroutine Decls ⟩ end when the words `END DECLARE` are reached.
- `FORTRAN` blocks containing ONLY `FORTRAN` type declarations can also appear inside a declaration statement (See Chapter 6, section 6.8). It is the user's responsibility to place ONLY `FORTRAN` type declarations within `FORTRAN` blocks in the declaration section. The SOL compiler **DOES NOT** catch the error of using other `FORTRAN` statements in a declaration section `FORTRAN` block .

The following restrictions apply to the declaration statement, when it appears:

1) The word `DECLARE` must appear alone on a line.

2) Only blank lines and comment lines can separate the declaration section from the program header.

3) The words, `END DECLARE`, must appear together, alone on a single line.

Examples of the possible formats for the start of a SOL program follow:

i.
```
PROGRAM just_an_example
DECLARE
END DECLARE
   ! empty declaration section
```

ii.
```
PROGRAM just_an_example
   ! no declaration section
```

iii.
```
PROGRAM just_an_example
DECLARE
REAL a
LOGICAL s, t, e, f
END DECLARE
   ! declaration section with type declarations
```

This chapter is divided into three sections:

5.1 Discusses Variable type declarations.

5.2 Discusses Subroutine declarations.

5.3 Discusses the declaration section in subroutines, and how it differs from the declaration section in the main program. More information on a subroutine's declaration section can be found in Chapter 9.

5.1 VARIABLE TYPE DECLARATIONS

A variable type declaration associates a variable with a type. This is not to be confused with variable initialization, which associates a variable with a value for the first time.

- See Chapter 2, section 2.1.2 for more information on variable initialization.
- Variable type declarations can ONLY appear in the declaration sections of the main program and/or subroutines.
- Variables are always local to the routine (i.e. the main program or subroutine) in which they are declared.
- See Chapter 5, section 5.3 for details on subroutine declarations.

A variable declaration has the following syntax:

⟨ type ⟩ ⟨ variable list ⟩

where:

⟨ type ⟩ is one of the following:

1) INTEGER
2) LOGICAL
3) REAL

⟨ variable list ⟩ is one of the following:

1) a variable
2) a series of variables separated by commas.

- If you do not explicitly give a variable a type with a variable declaration, it is assumed to be of type **REAL**. Chapter 3 discusses types in detail.
- The order of the declaration is unimportant; you can make variable declarations in any order you wish.

The following restrictions apply to how variables can be declared:

1) The list of variables that follows the ⟨ type ⟩ cannot run over onto the next line, unless the next line is explicitly defined as a continuation line with an & symbol. (Remember, the & symbol must appear in the first column, See Chapter 2, section 2.4 for details). Some examples follow:

Legal	*Illegal*
`LOGICAL a, indexer,` `& found, b`	`LOGICAL a, indexer,` `found, b`

2) No more than one declaration can appear per line. For example:

Legal	*Illegal*
`LOGICAL I_m` `REAL legal`	`LOGICAL I_m REAL illegal`

3) A variable cannot be declared to be more than one type. So, for example:

Legal	*Illegal*
`LOGICAL a_namee` `INTEGER different_name`	`LOGICAL same_name` `INTEGER same_name`

4) You must put one or more blanks after the type name to distinguish it from the variable. For example:

```
LOGICALa_variable
```

is illegal.

5.2 SUBROUTINE DECLARATIONS

Subroutine declarations appear in the declaration section of the main program; the subroutine's name and its parameters are declared.

- A subroutine must be declared before it is called.

- Subroutine declarations can only appear in the declaration section of the main program.
- A subroutine declaration differs from a subroutine implementation.
- A subroutine declaration is a template describing the dependent and independent parameters, and the subroutine's name, although a subroutine implementation describes the specific action a subroutine will perform. A subroutine declaration provides no information about the specific action a subroutine will perform.
- Chapter 9 discusses subroutines in detail.

A subroutine declaration has the following syntax:

`SUBROUTINE` (⟨ dep params ⟩) = ⟨ sub name ⟩ (⟨ indep params ⟩)

where:

⟨ dep params ⟩ is the dependent parameter list. A parameter list is one of the following:

1) nothing, an empty list is possible
2) a single SOL identifier. (Assumed to be type `REAL`)
3) an identifier/type pair which consists of:

   ⟨ identifier ⟩ : ⟨ type ⟩
   where: ⟨ type ⟩ is `REAL, INTEGER or LOGICAL.`

4) A series of 2) and/or 3), separated by commas

For example, the following are legal parameter lists:

```
a, b, c, d :  REAL, e :  LOGICAL, f
a
a_var_param :  INTEGER
```

⟨ sub name ⟩ is a SOL identifier representing the subroutine's name.

⟨ indep params ⟩ is the independent parameter list, and has a syntax identical to the ⟨ dep params ⟩ stated previously.

The following restrictions apply to subroutine declarations:

1) The entire subroutine declaration must appear on a single line. If it will not fit, the continuation symbol, &, must be used. See Chapter 2, section 2.4 which provides more information on the continuation symbol. For example:

Legal

```
SUBROUTINE
& (spread, out) = the_sub ()
```

Illegal

```
SUBROUTINE
(spread, out) = the_sub ()
```

2) Any subroutine called by the main program, or another subroutine, must be declared in the declaration section of the MAIN program.

3) Subroutine names must be unique or an error results. You cannot declare two subroutines with the same name.

4) Variables or parameters CANNOT have the same name as a subroutine.

5) Two parameters, in the same subroutine declaration, CANNOT have the same name.

6) Dependent parameters are given to the left of the equal sign and are altered by the subroutine.

7) Independent parameters appear to the right of the subroutine name, and are NOT altered by the subroutine.

EXAMPLES:

In the following example, each subroutine declaration is separated by a blank SOL comment, and the declaration section of a SOL main program is also shown:

```
DECLARE
   SUBROUTINE (dep, another : LOGICAL) = subrout
&  (an_indep , another_indep )
!
   SUBROUTINE () = a_sub (single :INTEGER)
!
   SuBroutine (dep : LOGICAL, another :REAL) = the_sub ()
!
   subroutine () = simple ()
END DECLARE
```

1) The first declaration declares the subroutine **subrout** with four parameters. The second dependent parameter, **another**, is explicitly declared to be of type **LOGICAL** whereas the remaining dependent parameter, **dep** and the independent parameters are implicitly assumed to be of type **REAL** since no type declaration appears.

2) The second declaration declares the subroutine **a_sub** with a single independent parameter of type **INTEGER**.

3) The third declaration declares the subroutine **the_sub** with two dependent parameters and no independent parameters.

4) The final declaration declares the subroutine **simple** with no parameters.

- Chapter 9 provides detailed information on subroutines, and their declaration.

5.3 THE DECLARATION SECTION IN SUBROUTINES

A declaration section in a subroutine is identical to the main program declaration section with one exception:

- No subroutines can be declared in the declaration section of a subroutine.

Otherwise main program and subroutine declaration sections are identical. The syntax and action described in section 5.1 of this chapter apply equally to both the main program and subroutine declaration sections.

- Chapter 9 discusses subroutines in detail.

Chapter 6
Statements

Following an overview of SOL statements, the chapter is divided into the following sections:

6.1 - Discusses the assignment statement

6.2 - Discusses the **PRINT** statements

6.3 - Discusses the conditional statement, **IF/THEN/ELSE**

6.4 - Discusses the repetitive statements

6.5 - Discusses the **ASSEMBLAGE** and **COMPONENT** statements

6.6 - Discusses the **OPTIMIZE** statement

6.7 - Discusses the subroutine call statement

6.8 - Discusses the use of **FORTRAN** blocks

OVERVIEW

SOL statements control the actions performed by a SOL program. The following restrictions apply to how SOL statements can be placed in a SOL program:

1) SOL statements can only appear three places in a SOL program:
   - In the SOL statement section of the main program.
   - In the SOL statement section of a subroutine implementation.
   - Inside another SOL statement, (multi-line statement).
2) Blank lines are allowed on the lines before and after all statements.
3) Comments are allowed on the lines before and after all statements.
4) Comments are allowed on the same line as a SOL statement, after the statement.
5) Two statements CANNOT appear on the same line.

Several SOL statements are *multi-line statements*; so a single statement can be many lines long. Regardless of the number of lines a statement occupies, a statement must start alone on a line, and must end alone on a line. The following are multi-line statements:

1) The **OPTIMIZE** statement
2) The **COMPONENT** and **ASSEMBLAGE** statements

3) The conditional **IF/THEN/ELSE** statement

4) The repetitive **DO** statements

- If a statement is not a multi-line statement, and it won't fit on a single line, then the continuation symbol, &, must be used. See chapter 2, section 2.4 which describes the continuation symbol in detail.
- The general structure of a SOL program is detailed in Chapter 2, section 2.1.5.

6.1 THE ASSIGNMENT STATEMENT

Assignment statements give variables a value. Variables can be initialized by assignment statements. The assignment statement has the following syntax:

⟨ id ⟩ = ⟨ expr ⟩

where:

⟨ id ⟩ is a SOL identifier. (Extended identifiers used for **ASSEMBLAGE**s or **COMPONENT**s CANNOT be used; See Chapter 7, section 7.1.2 for details).

⟨ expr ⟩ is a SOL expression.

The expression on the right side of the assignment statement's equal sign is evaluated and the resulting value is assigned to the variable.

The following restrictions apply to assignment statements:

1) Assignment statements must appear alone on a line. If an assignment statement will not fit on a line, the continuation symbol must be used. (Further details appear in Chapter 2, section 2.4). For example:

Legal	*Illegal*
`a_var = (43 +` `& r ** 2)`	`a_var = (43 +` `r ** 2)`

There are two types of assignment statements in SOL:

6.1.1 ARITHMETIC ASSIGNMENTS

The arithmetic assignment assigns to the variable on the left of the equal sign, the value of the arithmetic expression on the right of the equal sign.

The arithmetic assignment statement has the following syntax:

⟨ a_variable ⟩ = ⟨ arith_expression ⟩

where:

⟨ a_variable ⟩	is a REAL or INTEGER variable
⟨ arith_expression ⟩	is an arithmetic expression. Chapter 4, section 4.1 details arithmetic expressions.

The following restrictions apply to arithmetic assignment statements:

1) The equal sign does not mean "is equal to," as in common mathematical usage. It means, "is replaced by." For example:

```
a = a + 1
```

This statement means, "replace the current value of the variable named 'a,' with the sum of a's current value and one."

2) All variables which appear on the right-hand side of an assignment statement MUST be initialized before their appearance.

3) The expression on the right-hand side must evaluate to a REAL or INTEGER value.

4) Variables on the left side of an assignment statement cannot be signed. (e.g. - a = a * 3 is illegal because "-a" cannot appear on the left-hand side of an assignment statement.

Type checking of arithmetic assignment statements is relaxed. In some cases, if the value of the evaluated expression is of a different type from the variable, the result is converted into a value of the variable's type.

The following table details how types are converted:

Table 6-1:

Var Type	*Expression Type*	*Expression converted to*
INTEGER	INTEGER	INTEGER
INTEGER	REAL	INTEGER, REAL is truncated
INTEGER	LOGICAL	error — illegal assignment
REAL	INTEGER	REAL, INTEGER converted
REAL	REAL	REAL
REAL	LOGICAL	error - illegal assignment

Chapter 3, especially sections 3.1 - 3.3, offers some detail on the concept of TYPE.

6.1.2 Logical Assignments

The logical assignment statement assigns to the variable on the left of the equal sign, the value of the logical expression to the right of the equal sign.

The logical assignment statement has the following syntax:

⟨ a_var ⟩ = ⟨ logic_expr ⟩

where:

⟨ a_var ⟩ is a LOGICAL variable

⟨ logic_expr ⟩ is a logical expression. Chapter 4, section 1.2 details logical expressions.

The following restrictions apply to logical assignment statements:

1) The equal sign does not mean "is equal to," as in common mathematical usage. It means, "is replaced by." For example:

```
a = .not.  r
```

This statement means, "replace the current value of the LOGICAL variable named a, with the negation of the LOGICAL variable r."

2) All variables which appear in the right side of an assignment MUST be initialized before their appearance.

3) The expression on the right-hand side must evaluate to a LOGICAL value.

Type checking of logical assignment statements is strict: a LOGICAL variable can only be assigned a LOGICAL value. The following table details the results of such assignments:
Table 6-2:

Var Type	*Expression Type*	*Expression converted to*
LOGICAL	INTEGER	error — illegal assignment
LOGICAL	REAL	error — illegal assignment
LOGICAL	LOGICAL	LOGICAL, value of expression.

Chapter 3, especially sections 3.1 – 3.3, offers some detail on the concept of TYPE.

6.2 THE PRINT STATEMENTS

SOL print statements are used to write values and messages from your executing SOL program to an external logical unit, which by default is your terminal.

- The external unit is not user defined.
- SOL uses external logical unit 6 (SYS$OUTPUT on VAX/VMS systems).
- For interactive SOL programs external logical unit 6 will be your terminal by default.
- For batch jobs, this external logical unit 6 defaults to be the batch log file.

SOL provides two kinds of print statements:

1) the PRINT statement

2) the SUMMARIZE print statement (ADVANCED MATERIAL). The SUMMARIZE print statement is discussed in Chapter 7, section 7.1.1.3.

- The PRINT statement is general-purpose, compared to the SUMMARIZE print statement which is used to print the summarization variables of ASSEMBLAGEs or COMPONENTs.
- See Chapter 7 for more detailed information on ASSEMBLAGEs or COMPONENTs.

This section is divided into two sections:

6.2.1 – Discusses the PRINT statement.

6.2.2 – Discusses the format part of PRINT and SUMMARIZE print statements.

6.2.1 PRINT STATEMENT

SOL's PRINT statement provides a means of writing values or messages from an executing SOL program to an external logical unit.

PRINT statements have the following syntax:

PRINT ⟨ print_list ⟩

where:

⟨ print_list ⟩ is the list of items to be printed, consisting of one of the following:

1) a variable

Example: `PRINT a_variable`

2) a string

Example: `PRINT 'This is a string'`

- a string CANNOT be longer than 61 characters.

3) an optional sign(+ or -), followed by a number

Examples: `PRINT 23` or `PRINT -143.2`

4) a variable, followed by a colon, :, followed by the desired print format.

Example: `PRINT a_variable` : ⟨ format ⟩

5) an optional sign-number pair, followed by a colon, :, followed by a format.

Example: `PRINT -23.4` : ⟨ format ⟩

6) a mixed series of the five choices above, separated by commas.

Examples:

i) `PRINT a_var, -2.3` : ⟨ format ⟩

ii) `PRINT 'try Again', a_var` : ⟨ format ⟩

7) nothing

Example: **PRINT**

- This has the effect of writing a blank line.

In the examples above, the word, ⟨ format ⟩ , appears where an actual format would appear. Formats are discussed in the next section of this chapter, section 6.2.2.

The following restrictions apply to **PRINT** statements:

1) Variables used in **PRINT** statements must be initialized before use.

2) No comma appears after the last item in a print list.

3) The format MUST be compatible with the variable or number printed. See section 6.2.2 of this chapter for details.

4) Print lists must appear on the same line as the word, **PRINT**.

   - If the print list is too long to fit on the line, use the continuation symbol, &, to continue the list on the next line. See Chapter 2, section 2.4 for details on the continuation symbol.
   - A string cannot be "split" over two lines with a continuation symbol, rather the entire string must be moved to the next line.

5) Print lists CANNOT be longer than 20 items in length.

6.2.2 FORMATS FOR PRINT AND SUMMARIZE PRINT STATEMENTS

The **PRINT** and **SUMMARIZE** print statements use formats identically. The format part of either print statement specifies the way in which output data will be displayed.

There are four formats available in SOL

- The format must be compatible with the variable or number being printed, the compatibility rules are given above, or an error will result.
- Since summarization variables are of type **REAL** by default (Chapter 7, section 7.1.1.1), only the **E** and **F** formats are legal for use with the **SUMMARIZE** print statement.

6.2.2.1 The E Format

The E format is compatible with values of type **REAL**.

The E format has the following syntax:

E ⟨ width ⟩ . ⟨ digits ⟩

where:

⟨ width ⟩	is a positive integer representing the size of the print field.
⟨ digits ⟩	is a positive integer, representing the significant digits for rounding.

Note: No spaces can appear between the ⟨ width ⟩ , decimal point and ⟨ digits ⟩ .

EXAMPLES:

`E12.3, E 12.3, E 3.2, E12.12,` and `e12.7.`

The E format transfers the value to be printed, rounded to ⟨ digits ⟩ decimal digits and right-justified, to an external field that is ⟨ width ⟩ characters wide. The value is displayed with scientific notation.

The following restrictions apply:

1) If the value does not fill the field, leading blank spaces are inserted.

2) If the value is too large for the field, the entire field is filled with asterisks, and a runtime error message appears.

3) The term, ⟨ width ⟩ , should be at least ⟨ digits ⟩ + 7.

4) The term, ⟨ digits ⟩ , must be greater than zero. A runtime error results if the term ⟨ digits ⟩ equals zero.

5) The term, ⟨ width ⟩ , must be greater than zero, or a compile time error results.

6) Both of the terms, ⟨ width ⟩ and ⟨ digits ⟩ , must appear and must be separated by a decimal point. If a single term appears, or the decimal point is excluded, a compile-time error will result

Examples of the E format in action are given in the following table:

Table 6–3: E Format Output Example

• in what follows, ␣ means a blank space.

Format	*Internal value*	*Value printed*
E9.2	475867.222	␣ 0.48E+06
E12.5	475867.222	␣ 0.47587E+06
E12.3	0.00069	␣ ␣ ␣ 0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E12.0	456.777	runtime error (1)
E9	73.34	compile time error (2)
E.4	+1999.334	compile time error (3)
E12.5	6	compile time error (4)
E0.2	1.23456	compile time error (5)

1) The first runtime error occurs because the term, ⟨ digits ⟩ , equals zero.

2) Using E9 is illegal because no term ⟨ digits ⟩ , is given.

3) Using E.4 is illegal because no term ⟨ width ⟩ , is given.

4) Using E12.5 is illegal because an E format cannot be used with INTEGERs.

5) Using E0.2 is illegal because the term ⟨ width ⟩ , equals zero.

6.2.2.2 *The F Format*

The F format is compatible with values of type REAL. It has the following syntax:

F ⟨ width ⟩ . ⟨ digits ⟩

where:

⟨ width ⟩ is a positive integer representing the size of the print field.

⟨ digits ⟩ is a positive integer, representing the significant digits for rounding.

Note: No spaces can appear between the ⟨ width ⟩ , decimal point and ⟨ digits ⟩ .

EXAMPLES:

F12.3, F 12.3, F 3.2, F12.11, and f12.7.

The F format transfers the value to be printed, with the fractional part rounded to ⟨ digits ⟩ decimal digits and right-justified, to an external field that is ⟨ width ⟩ characters wide. The value is displayed with ordinary decimal notation.

The following restrictions apply:

1) If the value does not fill the field, leading blank spaces are inserted.

2) If the value is too large for the field, the entire field is filled with asterisks, and a runtime error message appears.

3) The term, ⟨ width ⟩ , should be at least ⟨ digits ⟩ + 4.

4) Both of the terms, ⟨ width ⟩ and ⟨ digits ⟩ , must appear and must be separated by a decimal point. If a single term appears, or the decimal point is excluded, an error will result.

Examples of the F format in action are given in the following table:

Table 6–4: F Format Output Example

• in what follows, ␣ means a blank space.

Format	*Internal value*	*Value printed*
F8.5	123456789	compile time error (1)
F8.5	-1234.567	******
F9.3	8789.7361	␣ 8789.736
F2.1	51.44	**
F10.4	-23.24352	␣ ␣ -23.2435
F11	93.45678	compile time error (2)
F.12	123.23178	compile time error (3)
F1.10	1234.34	*
F0.2	12.345987	compile time error (4)
F6.0	1234.567	␣ 1235. (rounded up)

1) The first compile time error for F8.5 occurs because the F format is not compatible with integer values, like 123456789.

2) The use of F11 is illegal because no term ⟨ digits ⟩ appears.

3) The use of F.12 is illegal because no term ⟨ width ⟩ appears.

4) The use of F0.2 is illegal because ⟨ width ⟩ must be greater than zero.

6.2.2.3 *The I Format*

The I format is compatible with values of type INTEGER. It has the following syntax:

I ⟨ width ⟩

where:

⟨ width ⟩ is a positive integer representing the size of the print field

EXAMPLES:

I12, I 9, I 3, I11, i 6, i, and I.

The I format transfers the value to be printed, right-justified, to an external field that is ⟨ width ⟩ characters wide. When ⟨ width ⟩ is left blank, a default value of twelve is used. The value is displayed with ordinary decimal notation.

The following restrictions apply:

1) **INTEGER** values cannot be larger than 2147483647. Attempting to display such a value will give a runtime error.

2) If the value does not fill the field, leading blank spaces are inserted.

3) If the value is too large for the field, the entire field is filled with asterisks, and a runtime error message appears.

4) The term, ⟨ width ⟩ , must be large enough to include a minus sign when necessary. (negative numbers will have minus sign as the leftmost nonblank character)

5) The term, ⟨ width ⟩ , cannot be zero.

Examples of the I format in action are given in the following table:

Table 6–5: I Format Output Example

- in what follows, ␣ means a blank space

Format	*Internal value*	*Value printed*
I3	284	284
I4	-284	-284
I5	174	␣ ␣ 174
I	123456	␣ ␣ ␣ ␣ ␣ ␣ ␣ 123456
I2	3244	**
I3	-473	***
I7	29.876	compile time error (1)
I	123456789012	compile time error (2)
I0	385	compile time error (3)

1) The use of I7 is an error because the I format is not compatible with a **REAL** value, like 29.876.

2) The use of I is not illegal, but the number 123456789012 is too large to be a legal integer, so a compile time error results. If the value is not known at compile-time, as with a variable, a runtime error would result instead.

3) The use of I0 is an error because the term ⟨ width ⟩ , equals zero.

6.2.2.4 The L Format

The L format is compatible with values of type **INTEGER** or type **LOGICAL**.

The L format has the following syntax:

L ⟨ width ⟩

where:

⟨ width ⟩	is a positive integer representing the size of the print field

EXAMPLES:

```
L12, L 9, L 3, L11, l 6, l, and L.
```

For LOGICAL values, the L format transfers either a T (if the value is `.true.`) or a F (if the value is `.false.`) to an external field that is ⟨ width ⟩ characters long. When ⟨ width ⟩ is left blank, a default value of two is used. The T or F is in the rightmost position of the field, preceded by ⟨ width ⟩ -1 spaces.

For INTEGER values, the L format transfers either a T (if the value equals one) or a F (if the value does not equal one). If ⟨ width ⟩ is left blank, a default value of two is used. Otherwise, the T or F is in the rightmost position of the field, preceded by ⟨ width ⟩ -1 spaces.

The following restrictions apply:

1) If the value does not fill the field, leading blank spaces are inserted.

2) The term, ⟨ width ⟩ , cannot be zero.

Examples of the L format in action are given in the following table:

Table 6–6: L Format Output Example

• in what follows, ␣ means a blank space

Format	*Internal value*	*Value printed*
L	.true.	␣ T
L1	.true.	T
L4	.false.	␣ ␣ ␣ F
L	28	␣ F
L1	1	T
L0	.false.	compile time error
L2	1.0	compile time error

1) The use of the L0 format is illegal because ⟨ width ⟩ equals zero.

2) The use of the L2 format is illegal because the L format cannot be used with REAL values, like 1.0.

SOL provides a single (multi-line) statement for conditional branching: the **IF** statement. The **IF** statement consists of two parts:

1) a required **THEN** part

2) an optional **ELSE** part

- The **IF** statement evaluates a logical expression and performs a specified action, the **THEN** part, if the expression evaluates to `.true.`
- The **ELSE** part is optional. When present, it only executes if the logical expression evaluates to `.false.`

The **IF** statement has the following syntax:

```
IF ⟨ logic_expr ⟩ THEN
    ⟨ SOL_statements ⟩
⟨ optional_ELSE ⟩
END IF
```

where:

⟨ logic_expr ⟩	is a logical expression. See Chapter 4, section 4.2.
⟨ SOL_statements ⟩	is a series of one or more SOL statements.
⟨ Optional_ELSE ⟩	is one of the following: 1) Nothing — the **ELSE** part is optional. 2) An **ELSE** part, with the following syntax: **ELSE** ⟨ SOL_statements ⟩ where : ⟨ SOL_statements ⟩ is a series of zero or more SOL statements.

So, an **IF** statement will have one of the following formats:

1)

```
IF ⟨ logic_expr ⟩ THEN
   ⟨ SOL_statements ⟩
END IF
```

2)

```
IF ⟨ logic_expr ⟩ THEN
   ⟨ SOL_statements ⟩
ELSE
   ⟨ SOL_statements ⟩
END IF
```

- Additionally, you can use **ENDIF**, with no space, as a replacement for **END IF**

The following restrictions apply to IF statements:

1) The word IF, the (logic expr) , and the word THEN MUST appear alone on a single line. If they will not fit on a single line, then the continuation symbol, &, MUST be used. Further details are in Chapter 2, section 2.4.

Legal

```
IF .not.
& a THEN
PRINT 'I am ok'
ENDIF
```

Illegal

```
IF .not.
a THEN
PRINT 'I am not ok'
ENDIF
```

2) The word ELSE, when it appears, MUST be alone on a single line.

Legal

```
IF a .gt.  10 THEN
PRINT 'This is ok'
ELSE
PRINT 'Ok too'
ENDIF
```

Illegal

```
IF a .gt.  10 THEN
PRINT 'The then part is ok'
ELSE print 'But the else part is illegal'
ENDIF
```

3) The ENDIF or END IF MUST appear alone on a line.

4) IF statements can be nested. The word ENDIF will match the nearest preceding IF. For example:

```
01 IF a .lt. 20 THEN
02   IF a .gt. 5 THEN
03     PRINT 'greater than 5'
04   ELSE
05     IF a .eq. 5 THEN
06       PRINT 'equals 5'
07     ENDIF
08 ENDIF
```

- The ENDIF on line 7 matches the IF on line 5.
- The ENDIF on line 8 matches the IF on line 2.
- An error results because there is no ENDIF for the IF statement on line 1.
- Disregard indentation when matching ENDIFs.

5) The IF/THEN/ELSE statement is a block in SOL, and must abide by special scope rules outlined in section 6.3.1 of this chapter.

6) ASSEMBLAGE or COMPONENT statements CANNOT appear inside an IF statement.

EXAMPLES:

Example 1:

```
IF crazy THEN
  print 'Yep, you are crazy'
ELSE
  print 'Nope, you OK'
END IF
```

This example prints a different line of text, depending on the value of the logical variable, `crazy`.

Example 2:

```
IF (a .gt. b) .AND.
&  (b .gt. c) THEN
  PRINT 'a is greater than c '
desirability = 50
ELSE
  IF (a.lt. b) .AND.
&     (b .lt. c ) THEN
     PRINT 'a is less than c '
     desirability = 0
   ELSE
     PRINT 'What do I know?'
     desirability = 100
   ENDIF
END IF
```

This example prints a different line of text, and sets the value of a variable, `desirability`, depending on the values of two relational expressions. It also illustrates the use of the & continuation symbol to continue a long expression (See Chapter 2, section 2.4).

6.3.1 SCOPE RULES FOR IF STATEMENTS

The `IF` statement is a block in SOL, which affects how variables are initialized within an `IF` statement:

- The chief characteristic of a block is that variables can be initialized within the block and remain uninitialized outside the block. Variables with this property are called *local* variables.
- It is possible to initialize local variables in the `THEN` part, and/or the `ELSE` part of an `IF` statement.
- Variable initializations are considered local to an `IF` statement if and only if both of the following are true:
  - i. The variable is uninitialized at the start of the `IF` statement.

ii. The variable is initialized in either the THEN part of the IF statement, or the ELSE part, but NOT BOTH. A variable which is initialized in both the THEN and the ELSE part of an IF statement is NOT considered local to the IF statement.

- When the IF statement ends, indicated by the words ENDIF or END IF, local variables initialized within the IF statement become uninitialized.
- In Chapter 2, section 2.1.2 the concepts of scope, block, and variable initialization are discussed.
- Chapter 11 discusses scope rules in depth.

These scope rules follow from SOL's stringent error checking. One cannot be certain that the statements in the THEN or the ELSE portion of an IF statement will be executed. Statements in the THEN part are executed only when the condition is true, and statements in the ELSE part are executed only when the condition is false. Therefore, it is uncertain whether a variable initialization which only appears in either the THEN or ELSE portion of an IF statement, will occur. SOL will not allow a variable to be used unless its initialization is certain. Hence, the scope rules require that only a local variable be initialized when it is not certain that the initialization will actually take place.

EXAMPLES:

In these examples, assume all variables are uninitialized before the code in the example begins.

Example 1:

```
a = 65
IF a .gt. 30 THEN
  a = 7
ENDIF
```

Is variable, "a," local to the IF statement?

NO. The variable was initialized prior to the start of the IF statement. After the IF statement, "a" will have the value 7.

Example 2:

```
IF .true. THEN
   a = 6
   b = 4
ELSE
   b = 9
ENDIF
```

Is variable, "a," local to the IF statement?

YES. It is uninitialized at the start of the IF, and "a" is initialized in the THEN part only. After the IF statement ends, "a" will be uninitialized, and cannot be printed, used in arithmetic expressions and so on.

Is variable, "b," local to the IF statement?

NO. It is initialized in both the THEN and the ELSE part of the IF statement. The variable "b" will have the value 4 after the IF statement has ended (because .true. is always true).

Example 3:

```
 c = 20
IF c .gt. 2 THEN
   a = 6
ELSE
   a = 7
ENDIF

IF a .lt. 12 THEN
    a = 10
ENDIF
```

Is variable, "a," local to the second IF statement?

NO. Because the variable "a" was initialized in both the THEN and ELSE part of the first IF, "a" is not local to the first IF. Thus, "a" has been initialized BEFORE the start of the second IF statement, and CANNOT be local to the second IF. When the second IF statement ends, "a" will have the value 10.

Example 4:

```
IF .true. THEN
c = 20
   IF c .ge. 15 THEN
     a = 7
   ENDIF
ELSE
  a = 6
ENDIF
```

Is the variable, "a," local to the encompassing IF statement?

YES. "a" is only initialized in the ELSE part of the encompassing IF statement, so it is a local variable. In the THEN part of the encompassing IF statement, another IF statement appears. This IF statement initializes a *local* variable, "a." This local "a" becomes uninitialized when the inner IF statement ends, so "a" is never initialized in the THEN part of the encompassing IF.

Example 5:

```
c = 5
IF c .lt. 12 THEN
   IF c .gt. 10 THEN
     a = 7
   ELSE
     a = 4
   ENDIF
ELSE
  a = 6
ENDIF
```

Is the variable, "a," local to the encompassing IF?

NO. "a" is initialized in both the THEN and ELSE parts of the encompassing IF, so "a" is not local. This example is similar to 4), except that the inner IF statement, inside the encompassing IF statement's THEN, initializes "a," in both the THEN and ELSE part, so that the variable "a" is initialized in the THEN part of the encompassing IF statement.

- The rules are simple, but when IF statements are nested within themselves and other statements, applying the rules may become a little complicated.
- These rules exist to prevent key variables from accidently being left uninitialized.
- A variable that is uninitialized before an IF statement, can ONLY be initialized when the variable is initialized in both the THEN and in the ELSE part of the IF statement.
- The rules insure that if a variable initialization is dependent on a condition, only a local variable is initialized.

6.4 REPETITIVE STATEMENTS

SOL's repetitive control statements are called DO loops. DO loops specify the repetitive execution of one or more SOL statements. SOL provides two types of repetitive control statements:

6.4.1 THE ITERATIVE DO LOOP

The iterative DO loop specifies the repetitive execution of a statement or statements, based on the value of an automatically incremented control variable.

The iterative DO loop has the following syntax:

```
DO ⟨ var ⟩ = ⟨ initial_exp ⟩ , ⟨ final_exp ⟩
   ⟨ loop_body ⟩
END DO
```

where :

⟨ var ⟩ is a legal SOL identifier. This is called the loop control variable, because it controls the iteration of the loop.

⟨ initial_exp ⟩ is an arithmetic expression. See Chapter 4, section 4.1 for more information on arithmetic expressions.

⟨ final_exp ⟩ is an arithmetic expression. See Chapter 4, section 4.1 for more information on arithmetic expressions.

⟨ loop_body ⟩ is one or more SOL statements.

Additionally, the single word, ENDDO, can be used instead of the words, END DO.

Generally, an iterative loop works as follows:

1) The value of ⟨ initial_exp ⟩ is calculated.

2) The control variable is assigned the value of ⟨ initial_exp ⟩ .

3) The value of ⟨ final_exp ⟩ is calculated.

4) If the value of the control variable is less than or equal to the value of ⟨ final_exp ⟩ then SOL statements in ⟨ loop_body ⟩ are executed; the control variable is incremented by 1; and this step is repeated until the control variable is greater than the value of ⟨ final_exp ⟩ . The loop ends, skipping step 5), and program execution continues with the statements that follow the ENDDO.

5) If the INITIAL value of the control variable is greater than the value of ⟨ final_exp ⟩ , the loop **executes once and a runtime warning message appears**. The control variable is incremented, and then the loop ends. Program execution continues with the statements which follow the ENDDO.

- Because of rule 5, an iterative DO loop will **always** execute at least once.

EXAMPLES:

Example 1:

```
DO control = 1 , 5
  PRINT 'Here I go again'
ENDDO
PRINT control : F4.2
```

will produce the following output at the terminal:

```
Here I go again
Here I go again
Here I go again
Here I go again
Here I go again
6.00
```

The following restrictions apply to iterative DO loops:

1) The value of the control variable CANNOT be altered inside the loop. This restriction insures that the control variable is incremented with every iteration. If you attempt to do so, a compile-time error will result.

2) All variables which appear in the initial and final arithmetic expressions must be initialized prior to their use.

3) The control variable must be of type REAL or INTEGER.

4) The values of both arithmetic expressions will be treated as being of the same type as the control variable. Thus, if our control variable is of type **INTEGER**, the following loop:

```
DO control = 1.2, 10.9
   PRINT 'Convert me'
ENDDO
```

will be treated as if you typed the following:

```
DO control = 1, 10
   PRINT 'Convert me'
ENDDO
```

REAL values are truncated, not rounded, when converting from **REAL** to **INTEGER**.

5) The start of the loop, consisting of "**DO** ⟨ var ⟩ = ⟨ initial_exp ⟩ , ⟨ final_exp ⟩ ," must appear alone on a line. If it is too long to fit alone on a line, the continuation symbol, &, must be used. (Chapter 2, section 2.4 offers a detailed discussion of the continuation symbol)

6) The end of the do loop consisting of **ENDDO** or **END DO**, must appear alone on a line.

7) A **COMPONENT** or **ASSEMBLAGE** statement CANNOT appear inside a loop

8) An inner loop's control variable cannot have the same name as an outer loop's control variable.

Some helpful facts about iterative do loops:

1) The loop will eventually terminate. The initial value is calculated at the start of the loop, and cannot change. The final value is also calculated at the start and cannot change. Therefore, there is a finite difference between the initial and final value. As the control variable is incremented each time the loop repeats, eventually the control variable will be larger than the final value, and the loop will terminate.

2) If the initial value is $\geq$ the final value, the statements of ⟨ loop_body ⟩ will execute only once.

3) Loops can be nested inside each other. However, loops cannot alter the value of an enclosing loop's control variable or an error will result. Also, a loop's control variable cannot have the same name as an enclosing loop's control variable.

Example 1:

```
DO n = 1, 5
   print n : f4.2
ENDDO
print 'the final value: ',n : f4.2
```

This loop will produce the following output at the terminal:

```
1.00
2.00
3.00
4.00
5.00
THE FINAL VALUE: 6.00
```

Example 2:

```
DO n = 1, 3
   print n : f4.2
   DO another = 4, 6
     print '  ', another : f4.2
   ENDDO
ENDDO
print 'the final value: ',n : f4.2
```

This loop will produce the following output at the terminal:

```
1.00
  4.00
  5.00
  6.00
2.00
  4.00
  5.00
  6.00
3.00
  4.00
  5.00
  6.00
THE FINAL VALUE: 4.00
```

This example illustrates the effects of nested loops.

Example 3:

Line numbers have been displayed to facilitate discussion:

```
01  DO a = 12,1
02     print 'here I am'
03  ENDDO
04  print a : f5.2
```

This loop produces the following output at the terminal:

```
*** RUNTIME WARNING FOR LINE NUMBER: 1
DO A = 12, 1
INITIAL BOUND IS 12.00
FINAL BOUND IS 1.00
WARNING: INITIAL > FINAL *** LOOP WILL EXECUTE ONCE
here I am
13.00
```

A warning message, at runtime, is issued because the initial value is greater than the final value. The loop **executes** once.

6.4.2 The Conditional DO Loop

The conditional DO loop executes one or more statements until a specified condition is true. The conditional DO loop has the following syntax:

```
DO
   ⟨ loop_body ⟩
END DO WHEN ⟨ logical_exp ⟩
```

where:

⟨ loop_body ⟩ is one or more SOL statements

⟨ logical_exp ⟩ is a logical expression (one that evaluates to `.true.` or `.false.` See chapter 4 for details.)

Additionally, the single word, ENDDO, can be used instead of the words, END DO.
In general, the conditional DO loop works as follows:

1) The statements in ⟨ loop_body ⟩ are executed.

2) The value of ⟨ logical_exp ⟩ is calculated. If the ⟨ logical_exp ⟩ evaluates to `.true.`, the loop is terminated and program execution continues with the statements after the word, ENDDO.

3) Steps 1) through 2) are repeated until the loop terminates.

Clearly, if ⟨ Logical_exp ⟩ never evaluates to `.true.`, the DO loop will not terminate, and an infinite loop will result.

For example, consider the following conditional DO loop:

```
stop = .false.
n = 0
DO
  n = n + 1
  PRINT n : F4.2
  IF n .eq. 5 THEN
    stop = .true.
  END IF
ENDDO WHEN stop
```

This loop will produce the following output at the terminal:

```
1.00
2.00
3.00
4.00
5.00
```

The execution of the loop terminates when **stop** is assigned to be **.true.**, which occurs when n equals five.

The following restrictions apply to conditional DO loops:

1) The word, DO, must appear alone on a line, or an error will result.

2) The end of a loop, "**ENDDO WHEN** ⟨ logical_exp ⟩ ," must appear alone on a line. If your logical expression is too long to fit on a single line, the line must be continued with the continuation symbol, &. See Chapter 2, section 2.4 which provides more information on the use of the continuation symbol.

3) If ⟨ logical_exp ⟩ consists of a single variable, and that variable is left uninitialized or is missing, an error will result. (The SOL compiler will inform you that the missing variable has been replaced with the symbol, **.true.**)

4) Any variables used in the logical expression, ⟨ logical_exp ⟩ must be initialized prior to use. (See Chapter 4 which provides more information about logical expressions)

5) A COMPONENT or ASSEMBLAGE statement cannot appear inside a DO loop.

6.5 ASSEMBLAGE AND COMPONENT STATEMENTS

The ASSEMBLAGE statement is a data/modeling structure used for sizing. An ASSEMBLAGE models a "whole;" but an ASSEMBLAGE is a special kind of "whole," one which equals the sum of its parts. The COMPONENT definition statement represents an individual piece of the total ASSEMBLAGE. For example, an airplane wing can be considered as an ASSEMBLAGE, where the weight of the wing equals the sum of the COMPONENT parts, such as flaps, wing box, skin, hydraulic systems, cooling systems, engine mounts and so on. Because ASSEMBLAGEs and COMPONENTs are a unique feature of SOL, Chapter 7 is devoted entirely to a detailed discussion of ASSEMBLAGEs and COMPONENTs.

6.6 THE OPTIMIZE STATEMENT

The **OPTIMIZE** statement provides an interface into a state-of-the-art numerical optimization routine, ADS. Thus, optimization is a high-level statement in SOL. Chapter 8 is devoted to a detailed discussion of the **OPTIMIZE** statement. For further information about ADS proper, please consult "ADS — A FORTRAN PROGRAM FOR AUTOMATED DESIGN SYNTHESIS — VERSION 1.10", NASA Contractor Report 177985, Grant NAG1-567, 1983, by G.N. Vanderplaats.

6.7 THE SUBROUTINE CALL

A subroutine call specifies parameters that will be passed to a routine and executes the routine. SOL subroutine calls have the following syntax:

(⟨ dependent list ⟩) = ⟨ routine name ⟩ (⟨ independent list ⟩)

where:

⟨ dependent list ⟩ is the list of dependent parameters, the variables which will be initialized or altered by the subroutine. This list consists of one of the following:
1) an empty list, nothing.
2) a single variable
3) a series of variables, separated by commas.
• no comma can appear after the last item in the list.
More information on the dependent parameter list can be found in Chapter 9, section 9.2.

⟨ routine name ⟩ is the name of the subroutine. This must be a legal SOL identifier.

⟨ independent list ⟩ is the list of independent parameters, the variables which will supply data needed as input to the subroutine. An independent parameter list has the same syntax as the ⟨ dependent list ⟩ detailed above. More information on the ⟨ independent list ⟩ can be found in Chapter 9, section 9.2.

The following restrictions apply to Subroutine calls:

1) The subroutine call must appear on a single line in your SOL program. If the call will not fit, then the continuation symbol & must be used. (See Chapter 2, section 2.4 which details the use of the continuation symbol)

2) The subroutine MUST be declared in the main program declaration section before it is called. (See Chapter 5, section 5.2)

3) The subroutine MUST be implemented in the subroutine implementation section or an error will result.

4) The subroutine name MUST be the same for the declaration, call, and implementation. Further, the formal and actual parameters MUST match in both number and type. (See Chapter 9, section 9.2 for details).

5) Only variables can be passed as parameters, values such as: `6`, `8.00`, or `.true.` may not be passed directly as subroutine parameters.

6) Any variable passed as an independent parameter to a subroutine MUST be initialized before the subroutine call.

EXAMPLES:

Example 1:

```
(x, y) = Thoth()
```

This statement calls the subroutine named, `Thoth`, and returns the variables x and y as dependent parameters. There are no independent variables, so the independent parameter list is empty.

Example 2: `(x) = Calorific_Caluminations(x)`

This statement calls the subroutine named, `Calorific_Caluminations`, and passes x as an independent parameter and returns x as a dependent parameter.

Example 3:

```
() = Craven_Dastard ()
```

This statement is a call to subroutine `Craven_Dastard`, with neither dependent nor independent parameters.

Chapter 9 gives a detailed discussion of subroutines.

6.8 FORTRAN BLOCKS (ADVANCED MATERIAL)

You can write FORTRAN code inside a SOL program. The FORTRAN code must be delimited; with /* indicating the beginning of a FORTRAN block, and * indicating the end of a FORTRAN block. In this way, a SOL program can interface with existing FORTRAN routines.

There are several important restrictions on the use of FORTRAN blocks:

1) FORTRAN blocks can only appear in the statement and declaration sections of the main program or subroutine implementations.

2) FORTRAN type declarations should ONLY appear in a FORTRAN block INSIDE a SOL DECLARATION section. ONLY FORTRAN type declarations should appear in FORTRAN blocks inside a SOL declaration section.

   - The SOL compiler **does NOT offer error-checking** for the contents of FORTRAN blocks. It is the SOL user's responsibility to use FORTRAN blocks inside SOL declaration sections correctly.

3) ONLY FORTRAN statements should appear in FORTRAN blocks inside the statement sections of SOL programs.

- SOL does NO error checking on the FORTRAN code that appears within a FORTRAN block. Thus, if you make a FORTRAN coding mistake it will not be caught by the SOL compiler, but should be detected when the FORTRAN output of the SOL compiler is compiled using the FORTRAN compiler.

4) FORTRAN blocks cannot appear before the start of the main program header, or an error will result.

5) FORTRAN block delimiters, /* and *, MUST begin in column one of a SOL program or an error will result.

6) Macro calls SHOULD NOT be used inside a FORTRAN block — only FORTRAN code should appear in a FORTRAN block (See Appendix C). However, FORTRAN blocks can be part of a macro's replacement text.

7) FORTRAN blocks CANNOT appear inside an ASSEMBLAGE or COMPONENT statement.

In general, it is best to abide by the following guidelines:

- write each FORTRAN block delimiter on a line by itself, and write the FORTRAN code between the delimiters.
- Do not nest FORTRAN blocks inside of other SOL statements, such as DO loops or IF statements. Keep FORTRAN blocks at the main program or subroutine level.
- FORTRAN blocks should ONLY be used to access variables initialized in the main program or subroutine implementation BEFORE the FORTRAN block appears; local variables should not be accessed.
- Columns are significant in FORTRAN, so space carefully inside your FORTRAN blocks. Be sure to indent correctly inside the FORTRAN block.

Example 1:

```
PROGRAM ftn_block
! Notice the delimiters start in column one, and the FORTRAN
! block accesses previously initialized non-local variables only.
! This is the safest way to use FORTRAN blocks.  The FORTRAN code
! produced by the compiler will need to be "linked" by the
! programmer to the external subroutine RANDOM, called
! in what follows.
DECLARE
  INTEGER seed
END DECLARE
seed     = 1777
number = 0
/*
      CALL RANDOM(seed, number)
\*
print number
end ftn_block
```

Example 2:

```
PROGRAM ftn_block
! Notice that the FORTRAN block accesses a unitialized variable, ''number.''
! This is not a good way to use FORTRAN BLOCKS.  In fact,
! because of the way SOL's variables work, the
! variable ''number'' is not initialized inside the IF
! statement.
DECLARE
  INTEGER seed
END DECLARE

seed = 1777
IF (seed .gt. 0) THEN
/*
      CALL RANDOM(seed, number)
\*
ELSE
  number = 0
ENDIF
end ftn_block
```

FORTRAN blocks should be avoided if possible.

- SOL does NO ERROR CHECKING on FORTRAN code introduced by FORTRAN blocks.
- FORTRAN blocks SHOULD NOT be used to access local variables, or to initialize variables.

Thus, FORTRAN blocks are sufficiently dangerous to be off-limits for the novice. Experienced SOL users should utilize FORTRAN blocks only when necessary.

ADVANCED MATERIAL: HOW TO INTERFACE WITH A FORTRAN ROUTINE

There are two ways to interface SOL with a **FORTRAN** routine: use a call to an external subroutine, or put the body of the **FORTRAN** block inside a SOL subroutine implementation. Both of these methods are described in detail in the text that follows.

I. External FORTRAN Routines:

External FORTRAN routines are written and compiled separately, and then "linked" to the **FORTRAN** object code produced by the SOL compiler. The external routine is called from within a SOL program through the use of a **FORTRAN** call statement in a SOL **FORTRAN** block. The two example programs given earlier use this technique to call an external subroutine named RANDOM. Thus, to use an external subroutine in a SOL program, the following procedure should be employed:

1) Write the SOL program.

2) Make sure all SOL variables that are to be used or altered by the external **FORTRAN** routine are NON-LOCAL and INITIALIZED BEFORE THE **FORTRAN** block. (This is a very IMPORTANT step)

3) Invoke the **FORTRAN** routine in the SOL program, using a SOL **FORTRAN** block that contains a **FORTRAN** CALL statement.

4) Compile the SOL program, using the SOL compiler.

5) Link the compiled SOL program with the "Linksol" Command Procedure; the "Linksol" command procedure prompts for the names of any external **FORTRAN** subroutines. (See Chapter 1, section 1.2 for details).

- The external routines need to be compiled separately (creating an ".obj" files) BEFORE they can be linked with the compiled SOL program.

II. Using the body of a FORTRAN routine inside a SOL routine

This method is a little more complicated than the first technique, but requires less fussing with the linker, and has the advantage of keeping a SOL program self-contained, with all of the code in a single source file.

As the basis for discussion of this method, the following SOL program accomplishes the same task as the earlier examples, but does not use an external subroutine call.

Example 3:

```
PROGRAM ftn_block
! Need to declare the subroutine in the declaration section

DECLARE
  INTEGER seed
  SUBROUTINE (seed_out : INTEGER, number) = Random (seed_in : INTEGER)
END DECLARE

seed = 1777
! calling the routine as a SOL subroutine call
(seed, number) = Random(seed)
PRINT number

END ftn_block

! The subroutine implementation.  It is especially important
! to initialize all dependent variables BEFORE accessing them in a FORTRAN block

SUBROUTINE (seed_out : INTEGER, number) = Random (seed_in : INTEGER)
!  Generate a random number using the linear congruential Method, D. Lehmer (1949)
DECLARE
/*
      INTEGER Multiplier, Increment
      REAL*8 Modulus
\*
END DECLARE
number    = 0
seed_out = 0
/*
      Modulus    = 65536.0
      Multiplier = 25173
      Increment = 13849
      seed_out   = MOD( (multiplier*seed_in+Increment), INT(Modulus))
      number      = seed_out/Modulus
\*
END Random
```

Thus, take the following steps to include the body of a FORTRAN routine as a SOL routine:

1) Declare a SOL subroutine in the declaration section with the same parameters as required by the FORTRAN routine, use SOL subroutine calls to call the SOL subroutine, and include it in the subroutine implementation section.

2) Define the types for local FORTRAN variables using a FORTRAN block in a SOL Declaration section.

3) Initialize all dependent variables AFTER the declaration section, but BEFORE a second FORTRAN block (step 5)) containing the body of the FORTRAN routine. (VERY IMPORTANT)

4) Make sure types will agree with all **FORTRAN** statements :

- **REAL** is equivalent to **REAL*8**
- **INTEGER** is equivalent to **INTEGER**.
- **LOGICAL** is equivalent to **LOGICAL**.

5) Use a second **FORTRAN** block to include the body of a **FORTRAN** routine as the statement section of a SOL routine.

6) **Caution:** Make sure the **FORTRAN** block alters any dependent parameters, unless you want the values defined in 3) above to be returned.

7) END the SOL subroutine.

8) Run the SOL compiler and linker as normal, unless **FORTRAN** code from the block requires special linking for external subroutines and so forth.

III. Using The SOL Compiler Output as an Aid to FORTRAN Blocks:

Perhaps the easiest way to see the effects of a **FORTRAN** block is to examine the **FORTRAN** output from the SOL compiler. This is especially helpful to make sure that things appear in the correct column. However, making changes to the **FORTRAN** output is STRONGLY discouraged. It is better to make a change in the SOL source and recompile. In this way, the more readable SOL source code accurately reflects what occurs when the program runs.

Chapter 7
Sizing: Assemblages and Components

ASSEMBLAGE and COMPONENT statements facilitate *sizing*, a type of engineering systems modeling. In this manual, sizing is defined as the modeling of a system *as the simple sum of its parts* with respect to some special *summarization variables*. For example, an airplane can be modeled and *sized* for weight; we model the major parts of the airplane, its systems and structural components, along with the interaction between the parts. The model is constructed so that *the weight of the entire airplane can be determined by summing the weight of its parts*, the systems and structural weights. Likewise the weight of any part of the airplane can be determined by summing *its* parts. In this case, "weight" is considered a *simple summarization variable*. SOL aids the modeling of such "assemblages;" by allowing the user to create such models, and automatically computing the necessary summations. In addition to simple summarization variables, SOL also offers *expression summarization variable*s, discussed subsequently in section 7.1.1.1 of this chapter.

- The ASSEMBLAGE statement models the whole structure, such as the airplane above.
- The COMPONENT statement models subsystems and structures of an ASSEMBLAGE, such as the airplane's wings, fuselage and landing gear. By nesting COMPONENTs inside other COMPONENTs and inside the ASSEMBLAGE, the whole structure can be modeled and sized.
- Simple summarization variables are automatically summed by SOL to yield correct totals for parts and subsystems (represented by COMPONENTs), and the whole structure (represented by the ASSEMBLAGE).
    - The simple summarization variables for an ASSEMBLAGE or a COMPONENT which does not contain nested COMPONENTs MUST be explicitly initialized.
- Expression summarization variable values are also computed automatically by SOL for each COMPONENT or ASSEMBLAGE, but expression summarization variables are not summed like simple summarization variables.
    - Section 7.1.1.1 of this chapter details expression summarization variables.
- An *extended identifier notation* allows variables in ASSEMBLAGEs and COMPONENTs to be accessed. The notation distinguishes among various COMPONENTs so that, for example, the weight of a wing box is distinguished from the weight of an insulation system.

EXAMPLES:

The following code fragment models an airplane wing as consisting of structural and electrical system components. A single summarization variable, w, is declared to size the wing for weight. The values of the component summarization variables are summed automatically to yield the total weight of the wing. The last line prints the total weight (179000 + 5000 in this case) of the ASSEMBLAGE using extended identifier notation (Chapter 7, section 7.1.2).

```
ASSEMBLAGE Wing (0, ' ')
!   The Summarization Section appears below
SUMMARIZE
   w
END SUMMARIZE

!      The Structural Component of the wing
   COMPONENT Structure (1, ' ')
      w = 179000
   END Structure

!      The Electrical system Component of the wing
   COMPONENT Electrical (1, ' ')
      w = 5000
   END Electrical
END Wing
PRINT w@Wing
```

This chapter is divided into the following sections which detail ASSEMBLAGE and COMPONENT usage:

7.1 — Offers an overview of the ASSEMBLAGE and COMPONENT definition statements, addressing their syntax, simple and expression summarization variables and extended identifier notation.

7.2 — Scope rules for ASSEMBLAGE and COMPONENT statements.

7.3 — ADVANCED MATERIAL on ASSEMBLAGE and COMPONENT iteration.

7.1 ASSEMBLAGEs AND COMPONENTs

The syntax of ASSEMBLAGE and COMPONENT statements is nearly identical. There are only two differences:

- ASSEMBLAGEs have a ⟨ Summarization ⟩ Declaration section, where simple and expression summarization variables are declared. Since COMPONENT statements can ONLY appear inside an ASSEMBLAGE, COMPONENTs inherit their ASSEMBLAGE's summarization variables.
- COMPONENTs have an optional ⟨ Iteration ⟩ section, where iteration variables can be declared (See Chapter 7, section 7.3). Iteration variables for an ASSEMBLAGE appear inside the ASSEMBLAGE ⟨ Summarization ⟩ Declaration section, so a separate iteration section is not required.

The **ASSEMBLAGE** statement has the following syntax:

ASSEMBLAGE ⟨ name ⟩ (⟨ indentation ⟩ , ⟨ row label ⟩)
⟨ Summarization ⟩
⟨ Body ⟩
END ⟨ name ⟩

The **COMPONENT** statement has the following syntax:

COMPONENT ⟨ name ⟩ (⟨ indentation ⟩ , ⟨ row label ⟩)
⟨ Iterations ⟩
⟨ Body ⟩
END ⟨ name ⟩

where (for both **ASSEMBLAGE**s and **COMPONENT**s):

⟨ name ⟩ is a legal SOL identifier for the name of the **ASSEMBLAGE** or **COMPONENT**.

- Extended identifiers cannot be used as either **ASSEMBLAGE** or **COMPONENT** names.

⟨ indentation ⟩ is a legal SOL number or the word **TAB** followed by a legal SOL number. This provides information for **SUMMARIZE** print statements.

- See section 7.1.1.3 of this chapter for more information.

⟨ row label ⟩ is a legal SOL string, that consists of at least one character. Provides information for **SUMMARIZE** print statements.

- See section 7.1.1.3 of this chapter for more information.
- Note that the null string, '', is not permitted.

⟨ Summarization ⟩ declares simple and expression summarization variables (see Chapter 7, section 7.1.1) and iteration variables for the **ASSEMBLAGE** (see Chapter 7, sections 7.1.1 and 7.3).

⟨ Iterations ⟩ Iteration variables for **COMPONENT**s are declared in this OPTIONAL section (see Chapter 7, sections 7.1.1 and 7.3).

⟨ Body ⟩ is a series of one or more SOL statements.

- The use of **FORTRAN** Blocks inside an **ASSEMBLAGE** or **COMPONENT** definition is NOT allowed. See Chapter 6, section 6.8 for more details.
- **COMPONENT** statements can only appear inside of an **ASSEMBLAGE** or another **COMPONENT** statement.
- **ASSEMBLAGE** statments CANNOT appear inside another **ASSEMBLAGE** or **COMPONENT** statement.

I. RESTRICTIONS ON ASSEMBLAGEs AND COMPONENTs:

1) The restrictions listed above for an **ASSEMBLAGE** or **COMPONENT** ⟨ body ⟩ must be followed.

2) The ⟨ name ⟩ at the start of the ASSEMBLAGE or COMPONENT must be the same as the ⟨ name ⟩ used at the end of the ASSEMBLAGE or COMPONENT.

3) An ASSEMBLAGE or COMPONENT ⟨ name ⟩ cannot be an extended identifier.

4) The ASSEMBLAGE or COMPONENT header, consisting of the word, "ASSEMBLAGE" or "COMPONENT;" the ASSEMBLAGE or COMPONENT ⟨ name ⟩ ; and summarize print information enclosed in parenthesis, MUST appear alone on a line. If the entire ASSEMBLAGE or COMPONENT header will not fit on a single line, the continuation symbol, &, must be used. For further information, see Chapter 2, section 2.4.

5) An ASSEMBLAGE or COMPONENT definition statement CANNOT appear inside an IF statement, or a SOL error will result.

6) An ASSEMBLAGE or COMPONENT definition statement CANNOT appear inside a DO loop, or a SOL error will result.

7) ASSEMBLAGEs and COMPONENTs are SOL blocks, and abide by special scope rules which determine how variables are initialized or accessed. See Chapter 7, section 7.2 and Chapter 11.

8) Other restrictions concerning the use of summarization variables and iteration variables appear in Chapter 7, section 7.1.1 and Chapter 7, section 7.3 respectively.

II. RESTRICTIONS ON ASSEMBLAGEs ONLY:

1) At MOST ONE ASSEMBLAGE statement can appear in the body of the main program or a subroutine.

2) An ASSEMBLAGE statement CANNOT appear inside of a COMPONENT statement; COMPONENTs are parts of an ASSEMBLAGE and not vice-versa.

III. RESTRICTIONS ON COMPONENTs ONLY:

1) COMPONENT definition statements can ONLY appear inside of an ASSEMBLAGE or inside other COMPONENT definition statements.

2) COMPONENTs nested at the same level inside an ASSEMBLAGE or another COMPONENT CANNOT have the same name (or there could be no way of distinguishing between them).

Legal

```
ASSEMBLAGE a (0, ' ')
SUMMARIZE
  x
END SUMMARIZE
COMPONENT same (0, ' ')
  x = 2
END same
COMPONENT diff (0, ' ')
  x = 2
END diff
END a
```

Illegal

```
ASSEMBLAGE a (0, ' ')
SUMMARIZE
  x
END SUMMARIZE
COMPONENT same (0, ' ')
  x = 2
END same
COMPONENT same (0, ' ')
  x = 2
END same
END a
```

Each ASSEMBLAGE MUST have an associated set of *simple summarization variables*, which represent the sizing information (such as weight) associated with the ASSEMBLAGE. Each ASSEMBLAGE can also have an OPTIONAL associated set of *expression summarization variables*:

- Summarization variables are declared in the Summarization Declaration section of an ASSEMBLAGE (Described subsequently).
- Each COMPONENT that makes up the ASSEMBLAGE initializes local copies of the simple summarization variables (See Chapter 7, section 7.2 for scope rules).
- Expression summarization variables are initialized automatically when an ASSEMBLAGE or COMPONENT ends.
- If an ASSEMBLAGE or COMPONENT contains COMPONENTs nested within it, the "inner" COMPONENT's simple summarization variables are summed automatically to yield a total value for the "outer" ASSEMBLAGE or COMPONENT.
- If an ASSEMBLAGE or COMPONENT does NOT contain COMPONENTs nested within it, its simple summarization variables MUST be explicitly initialized (i.e. with assignment statements or subroutine calls).

The summarization declaration section of an ASSEMBLAGE has the following syntax:

```
SUMMARIZE
    ⟨ summarize decls ⟩
    ⟨ optional iterations ⟩
END SUMMARIZE
```

where:

⟨ summarize decls ⟩ is the set of summarization declarations. The summarization declarations end under either of the following conditions:

- The summarization declaration section ends with the words, END SUMMARIZE

OR

- if the ⟨ optional iterations ⟩ section has been supplied (See Chapter 7, section 7.3), the summarization declaration section ends when the word ITERATE appears.

⟨ optional iterations ⟩ is an optional iteration section where iteration variables can be declared (see Chapter 7, section 7.3).

A summarization declaration consists of one of the following:

1) A simple summarization variable declaration. (See Chapter 7, section 7.1.1.1 for exact syntax).
2) An expression summarization variable declaration (See Chapter 7, section 7.1.1.1 for exact syntax).
3) A summary title declaration, used in SUMMARIZE print statements. (See Chapter 7, sections 7.1.1.2 and 7.1.1.3 for details).
4) A series of 1) and/or 2) and/or 3) separated by commas OR carriage returns.

- **density** is an EXPRESSION summarization variable. The **density** of each COMPONENT is computed automatically using the expression declared with the variable, the **wt/vol** formula.
- The **density** of the total **Radio** is computed automatically using the the total **Radio** weight and volume (which were also computed automatically.

  Note: The two local **density** variables are NOT summed to yield the total **density** of the **Radio**.

Example 2: The following ASSEMBLAGE models a ballpoint pen sized for weight:

```
ASSEMBLAGE Pen (0, ' ')
SUMMARIZE
   weight
END SUMMARIZE
   COMPONENT Cap (1, ' ')
      weight = 1.322
   END Cap
   COMPONENT Shaft (1, ' ')
      COMPONENT Plastic (2, ' ')
         weight = 3.44
      END Plastic
      COMPONENT Nib_and_Ink
         weight = 2.786
      END Nib_and_Ink
   END Shaft
END Pen
```

- The **weight** of the **Shaft** is automatically given the sum of the **Plastic** and **Nib_and_Ink** COMPONENTs (6.226)
- The **weight** of the ASSEMBLAGE is similarily initialized with the sum of the **Shaft** and **Cap** COMPONENTs.

7.1.1.1 *Summarization Variable and Expression Variable Declarations: Syntax & Restrictions*

Simple summarization variables have the following syntax:

⟨ id ⟩

where:

⟨ id ⟩ is a SOL identifier.

Expression summarization variables have the following syntax:

⟨ id ⟩ = ⟨ arith expr ⟩

where:

⟨ id ⟩ is a SOL identifier, representing an expression summarization variable.

⟨ arith expr ⟩ is an arithmetic expression. See Chapter 4, section 4.1 for details.

The following restrictions apply to the summarization declaration section:

1) The word SUMMARIZE MUST appear alone on a line.

2) Comments and blank lines CAN appear between the ASSEMBLAGE header and the word SUMMARIZE.

3) Comments and blank lines CAN appear between the words SUMMARIZE and END SUMMARIZE.

4) The ⟨ summarize decls ⟩ part CANNOT be empty. At least one summarization variable declaration MUST appear.

The use of simple and expression summarization variables is best shown by example. Examples illustrating the use of the summarization declaration section are given next, followed by these three sections:

EXAMPLES:

Example 1: An ASSEMBLAGE with two simple summarization variables and an expression summarization variable:

```
ASSEMBLAGE Radio (0,  ' ')
SUMMARIZE
   wt,  vol
   density = wt/vol
END SUMMARIZE

  COMPONENT Housing (1, 'housing' )
     wt = 5000
     vol = 3000
  END Housing

  COMPONENT Knobs_n_Stuff (1, 'L.g' )
     wt = 0.03 * 10000
     vol = 30
  END Knobs_n_Stuff
END Radio
```

In this example, a radio is modeled as consisting of a housing component, with the knobs and electronics represented by a single component; the entire radio model is sized for weight and volume.

- wt and vol are SIMPLE summarization variables. The local wt and vol of the two COMPONENTs are summed to yield the total Radio weight and volume.

RESTRICTIONS FOR SUMMARIZATION DECLARATIONS

1) Summarization variables are ALWAYS LOCAL.

   - If a previously initialized variable is used as a summarization variable, a local copy is made instead and a warning message appears.

2) A summarization variable CANNOT be declared twice in the same declaration section.

3) All summarization variables must be of type REAL. Since SOL variables are REAL by default, no explicit declaration is needed.

4) EVERY summarization variable must be initialized in every ASSEMBLAGE or COMPONENT Statement according to the following rules:

   i. If COMPONENTs DO NOT appear in the body (See Chapter 7, section 7.1) of a given ASSEMBLAGE or COMPONENT, all simple summarization variables in that ASSEMBLAGE or COMPONENT MUST be explicitly initialized (e.g. via assignment statement or subroutine call).

   ii. If COMPONENTs DO appear in the body of a given ASSEMBLAGE or COMPONENT, all simple summarization variables in that ASSEMBLAGE or COMPONENT are automatically initialized and explicit initialization is ILLEGAL.

   iii. Expression summarization variables CANNOT be explicitly initialized under any circumstances, and are ALWAYS computed automatically when the ASSEMBLAGE or COMPONENT ends.

5) Since automatic initialization takes place when the COMPONENT or ASSEMBLAGE ends (cases ii. and iii. above) , automatically initialized variables CANNOT be accessed until after the COMPONENT or ASSEMBLAGE ends.

6) After a COMPONENT or ASSEMBLAGE ends, local variables such as summarization variables can be accessed (but NOT altered) via extended identifier notation (See Chapter 7, section 7.1.2).

7) Only two types of variables can be used in arithmetic expressions for summarization expression declarations:

   - Previously initialized variables.

Legal

```
some_var = 12
ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
a
b = some_var/2
END SUMMARIZE
```

Illegal

```
ASSEMBLAGE uninitalized (0, ' ')
SUMMARIZE
a
b = some_var/2
END SUMMARIZE
```

- Previously declared summarization variables.

Legal

```
ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
a
b = a/2
END SUMMARIZE
```

Illegal

```
ASSEMBLAGE not_declared_yet (0, ' '
SUMMARIZE
b = a/2
a
END SUMMARIZE
```

EXAMPLES:

Recall that summarization and summarization expression variables are declared in the summarization section of an ASSEMBLAGE. Both the ASSEMBLAGE header and summarization section are shown in the examples:

Example 1: Five simple summarization variables (a, b, c, d, e) and two expression summarization variables (f, g) are declared.

```
ASSEMBLAGE example (0, ' ')
SUMMARIZE
   a, b
   c
   d, e, f = a/b
   g = a + c - e + f
END SUMMARIZE
```

Example 2: A simple summarization variable, a, and an expression summarization variable, b, are declared.

```
PROGRAM Demo

ASSEMBLAGE trial (0, ' ')
SUMMARIZE
  a, b = a ** 2
END SUMMARIZE

 COMPONENT test (1, ' ')
    a = 5
  END test

  COMPONENT test_2 (1, ' ')
    a  = 6
  END test_2

  print  b@test : F3.0
END trial

print a@trial : F3.0
print b@trial : F3.0
END Demo
```

This program produces the following output:

```
25.
11.
121.
```

The following table lists the values for the summarization variables and explains how the values were computed.

Table 7–1
SSV = "Simple Summarization Variable"
ESV = "Expression Summarization Variable"

Var	*COMPONENT*	*Type*	*Value*	*How value was derived*
a	test	SSV	5	explicitly initialized since test has no subCOMPONENTs
a	test_2	SSV	6	explicitly initialized since test_2 has no subCOMPONENTs
a	trial	SSV	11	computed automatically by SOL; equals sum of the summarization variables for subCOMPONENTs, test and test_2. (5 + 6 = 11)
b	test	ESV	25	computed automatically by SOL; equals the summarization expression evaluated with local summarization variables. (a = 5, b = a ** 2 = 25)
b	test_2	ESV	36	computed automatically by SOL; equals the summarization expression evaluated with local summarization variables. (a = 6, b = a ** 2 = 36)
b	trial	ESV	121	computed automatically by SOL; equals the summarization expression evaluated with local summarization variables. (a = 11, b = a ** 2 = 121)

- The print statements use extended identifer notation. (See Chapter 7, section 7.1.2 for details).
- The symbol, @, should be read as "of the." For example, b@test should be read as, "variable b of the test component."
- The print statements will display the following values, in order from first to last: 25, 11, and 121, which represent "the value of variable b of the test component," "the value of variable a of the trial assemblage," and "the value of variable b of the trial assemblage" respectively.

Example 3: Two simple summarization variables, a and c, and an expression summarization variable b are declared. **Note:** this example illustrates the use of a global variable to declare an expression summarization variable.

```
PROGRAM Demo2
 global = 2
ASSEMBLAGE trial (0, ' ')
SUMMARIZE
  a, c, b = global ** 2
END SUMMARIZE

 COMPONENT test (1, ' ')
    a = 5
    c = 1
  END test
  COMPONENT test_2 (1, ' ')
    global = 3
    a  = 6
    c  = 2
  END test_2

  print b@test : F2.0
END trial

print a@test@trial : F2.0
print c@trial : F2.0
print b@trial : F2.0

END Demo2
```

This program produces the following output:

```
 4.
 5.
 3.
 9.
```

The following table lists the values for the summarization variables and explains how the values were computed.

Table 7-2
SSV = "Simple Summarization Variable"
ESV = "Expression Summarization Variable"

Var	*COMPONENT*	*Type*	*Value*	*How value was derived*
a	test	SSV	5	explicitly initialized since test has no subCOMPONENTs
a	test_2	SSV	6	explicitly initialized since test_2 has no sub COMPONENTs
a	trial	SSV	11	computed automatically by SOL; equals sum of the summarization variables for subCOMPONENTs, test and test_2. (5 + 6 = 11)
c	test	SSV	1	explicitly initialized since test is has no subCOMPONENTs
c	test_2	SSV	2	explicitly stated since test_2 has no subCOMPONENTs
c	trial	SSV	3	computed automatically by SOL; equals sum of the summarization variables for subCOMPONENTs, test and test_2. (5 + 6 = 11)
b	test	ESV	4	computed automatically by SOL; equals the summarization expression evaluated with the current value of global. (global = 2, b = global ** 2 = 4)
b	test_2	ESV	9	computed automatically by SOL; equals the summarization expression evaluated with the current value of global. (global = 3, b = global ** 2 = 9)
b	trial	ESV	9	computed automatically by SOL; equals the summarization expression evaluated with the current value of global. (global = 3, b = global ** 2 = 9)

- The print statements use extended identifer notation. (See Chapter 7, section 7.1.2 for details).
- The symbol, @, should be read as "of the." For example, b@test should be read as, "variable b of the test component."

- The print statements will display the following values, in order from first to last: 4, 5, 3, and 9, which represent "the value of variable `b` of the `test` component, "the value of variable `a` of the `test` component of the `trial` assemblage," "the value of variable `c` of the `trial` assemblage," and "the value of variable `b` of the `trial` assemblage" respectively.

Example 4: The example contains a compilation error because of a violation of rule 4 (iii.) of the restrictions on summarization variables. (Restrictions appear in this Chapter and section, just before the current "examples" section.)

```
PROGRAM Demo3

ASSEMBLAGE trial (0, ' ')
SUMMARIZE
  a
  b = a ** 2
END SUMMARIZE

 COMPONENT test (1, ' ')
    a = 5
    b = 14
  END test

  COMPONENT test_2 (1, ' ')
    a  = 6
  END test_2
END trial
END Demo3
```

- A compilation error occurs inside `COMPONENT test`. The statement, `b = 14`, is ILLEGAL because `b` is an expression summarization variable and CANNOT be explicitly initialized.

Example 5: This example contains a compilation error because of a violation of rule 5 of the restrictions on summarization variables. (Restrictions appear in this chapter and section, just before the current "examples" section).

```
PROGRAM Demo4

ASSEMBLAGE trial (0, ' ')
SUMMARIZE
  a
  b = a ** 2
END SUMMARIZE

  COMPONENT test (1, ' ')
    a = 5
  END test

  COMPONENT test_2 (1, ' ')
    a  = b * 6
  END test_2
  print b@test
END trial

END Demo4
```

- An error occurs inside `COMPONENT test_2`. The statement, `a = b * 6` is ILLEGAL, because the summarization expression variable `b` of `COMPONENT test_2` is not initialized until the `COMPONENT` ends.
- The summarization expression variable initialized in `COMPONENT test` is a local variable, and CANNOT be accessed from `COMPONENT test_2` unless extended identifier notation is used. (See section 7.1.2 for details on extended identifier notation).

7.1.1.2 Summary Title Declarations – Advanced Material

The `Summary_Title` declaration initializes a header title for `SUMMARIZE` PRINT statements.

- A `Summary_Title` declaration can appear in the Summarization declaration section of an `ASSEMBLAGE`.
- A `Summary_Title` declaration is OPTIONAL; it need NOT appear in the Summarization declaration section of an `ASSEMBLAGE`.
- More information on summarize print statements can be found in section 7.1.1.3.

A **Summary_Title** declaration has the following syntax:

SUMMARY_TITLE = ⟨ string ⟩

where:

⟨ string ⟩ is a SOL string, consisting of a string of characters enclosed by apostrophes.

- A string cannot be longer than 61 characters.
- The string must appear on a single line. The continuation symbol, &, CANNOT be used to split a string over two lines.

The following restrictions apply to **Summary_Title** declarations:

1) An empty (null) ⟨ string ⟩ such as '' is illegal.

2) **Summary_Title** declarations appear in the summarization declaration section of an **ASSEMBLAGE**, and must be separated from summarization variables by commas OR carriage returns. For example:

Legal

```
ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
  summary_title = 'Demo Title'
  w
END SUMMARIZE

ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
  summary_title = 'Demo2 Title', sum_var
END SUMMARIZE
```

Illegal

```
ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
  summary_title = 'Demo Title' w

END SUMMARIZE

ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
  summary_title = 'Demo2 Title' sum_var
END SUMMARIZE
```

3) The **Summary_Title** declaration must appear on a single line. If it will not fit, the continuation symbol, &, must be used. For further details, see Chapter 2, section 2.4. For example:

Legal

```
ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
  summary_title =
& 'use the continuation symbol'
  w, e = w * 2
END SUMMARIZE
```

Illegal

```
ASSEMBLAGE Demo (0, ' ')
SUMMARIZE
  summary_title =
'your decl is illegal without it'
  w, e = w * 2
END SUMMARIZE
```

Chapter 7, section 7.1.1.3 details the use of the summarize print statement.

An understanding of ASSEMBLAGEs and COMPONENTs, detailed earlier in this chapter, will make this section more understandable

- The SUMMARIZE print statement allows the quick, and concise printing of ASSEMBLAGE or COMPONENT summarization variables.
- The SUMMARIZE print statement prints summarization variables in a tabular form, along with a header message indicating whether the ASSEMBLAGE has ended.
- Each row in the table contains the values for a requested ASSEMBLAGE variable.
- Options allow a title for the table and/or row labels to be specified.
- Rows can be indented to reflect COMPONENT nesting within the ASSEMBLAGE, or for emphasis.

GENERAL USE AND SYNTAX OF THE SUMMARIZE PRINT STATEMENT

The SUMMARIZE print statement has the following syntax:

SUMMARIZE ⟨ print_list ⟩

where:

⟨ print_list ⟩ is the list of variables to be printed, and consists of one of the following:

1) a simple summarization variable, or expression summarization variable.

Example: `SUMMARIZE a_summarize_variable`

Recall that summarization variables must be declared in the summarize declaration section. (See chapter 7, sections 7.1.1 and 7.1.1.1 for greater detail on summarization variable declarations)

2) a simple summarization variable or expression summarization variable, followed by a colon, :, followed by a format.

Example: `SUMMARIZE a_sum_var :` ⟨ format ⟩

3) a mixed sequence of the two choices above, separated by commas.

Examples:

`SUMMARIZE a_var, another :` ⟨ format ⟩ `, a_third`

`SUMMARIZE a :` ⟨ format ⟩ `, b :` ⟨ format ⟩ `, last :` ⟨ format ⟩

- In the examples above, the word, ⟨ format ⟩ , appears where an actual format would appear. (Formats are discussed in Chapter 6, section 6.2.2).
- Since summarization variables are always of type REAL, only E and F formats are legal with SUMMARIZE print statements.

The following restrictions apply to SUMMARIZE print statements:

1) Variables used in SUMMARIZE print statements must be summarization variables.

2) No comma can appear after the last item in a print list.

3) You must use a format that is compatible with the variable printed. See Chapter 6, section 6.2.2 for details on formats.

4) Print lists must appear on the same line as the word, SUMMARIZE.

- If the print list is too long to fit on the line, use the continuation symbol, &, continue the list on the next line. See Chapter 2, section 2.4 for details on the continuation symbol.

5) Print lists can be NO LONGER than 20 items.

6) SUMMARIZE print statements are only legal if at least one ASSEMBLAGE or COMPONENT statement has ended before the SUMMARIZE print statement.

The following is an example of an ASSEMBLAGE statement, illustrating the use of the SUMMARIZE print:

```
01 PROGRAM example
02
03 ASSEMBLAGE One(0, 'the name one')
04 SUMMARIZE
05     a, b
06 END SUMMARIZE
07     COMPONENT Two(TAB 1, 'the name two')
08         a = 1
09         b = 2
10     END Two
11 COMPONENT Three(TAB 1, 'the name three')
12     a = 2
13     b = 3
14 END Three
15 END One
16 SUMMARIZE a :  f4.2, b :  f4.2
17 END example
```

The SUMMARIZE print statement on line sixteen results in the following output at the terminal:

```
SUMMARY STATEMENT

   NO TITLE DECLARED IN SUMMARY_TITLE
                          A          B
     THE NAME TWO      1.00       2.00
     THE NAME THREE    2.00       3.00
THE NAME ONE        3.00   5.00
```

The explanation of the example:

- The first thing printed is the table header, "SUMMARY STATEMENT ." If the SUMMARIZE print statement appeared before the ASSEMBLAGE had ended, the table header would have been, "PARTIAL SUMMARY STATEMENT. "
- A blank line appears next, which is then followed by the table title. As no title was declared, the default, "NO TITLE DECLARED IN SUMMARY_TITLE, appears. (See Chapter 7, section 7.1.1.2)

- The column headers appear next, automatically labeled with the names of the summarization variables that will be printed.
- The summarization variables for the first COMPONENT ended (COMPONENT ends at line 10) appear next.
- The row is labeled with the string declared in the COMPONENT definition, 'the **name two**' (Line 7). The label is idented one tab (five spaces). This was also specified by TAB which appears before the label in the COMPONENT definition (Line 7). The values of the variables follow.
- The indented label and the values for the next COMPONENT follow.
- Values of the ASSEMBLAGE summarization variables are then printed. This row is not indented as specified by the definition (on line 3, the number 0 specifies zero spaces). Because of the lack of indentation, the values for the summarization variables are not aligned with those of the previous two COMPONENTs.
- A title can be declared in the SUMMARIZE section of an ASSEMBLAGE (See section 7.1.1.2 of this chapter) that will be printed with the table of summarization variable values generated by a SUMMARIZE print.
  - If no title is explicitly declared, the default title will be used.
  - The default title is, NO TITLE DECLARED IN SUMMARY_TITLE.

LABELING AND IDENTING TABLE ROWS

Identation information must be given for the table rows produced by a SUMMARIZE print statement. A label for each row of a SUMMARIZE print table must also be specified.

- Each row of a SUMMARIZE print table states an indented label, followed by the values of the requested summarization variables for a particular ASSEMBLAGE or COMPONENT.
- Indentation and labeling instructions are given at the start of an ASSEMBLAGE or COMPONENT.

The start of an ASSEMBLAGE has the following syntax:

ASSEMBLAGE ⟨ name ⟩ (⟨ identation ⟩ , ⟨ a_string ⟩)

The start of a COMPONENT definition has the following syntax:

COMPONENT ⟨ name ⟩ (⟨ identation ⟩ , ⟨ a_string ⟩)

where (ASSEMBLAGEs and COMPONENTs) row labeling information is given by ⟨ a_string ⟩ and ⟨ indentation ⟩ :

⟨ name ⟩ is a legal SOL identifier which names the ASSEMBLAGE or COMPONENT.

⟨ indentation ⟩ is a legal SOL number or the word TAB followed by a legal SOL number.

- Specifies how many blank spaces to indent ⟨ a_string ⟩ .
- If the word TAB appears, the number specifies how many tabs (where a tab is five blank spaces) to indent.
- numbers less than or equal to -1 will result in the row not being included in the table.

- numbers are treated as INTEGERs and truncated. e.g. 1.9 becomes 1 and .999999 becomes zero. Only digits before the decimal point are significant.

⟨ a_string ⟩ is a legal, non-null SOL string.

- Row labeling information is given in ⟨ a_string ⟩ .
- The label specified in ⟨ a_string ⟩ will appear in the leftmost position of the table row.
- If no label is desired, use a space string, ' '. The null string, '' , is not allowed.

7.1.2 Extended Identifier Notation

Extended Identifier notation is used to access the values of variables initialized within an ASSEMBLAGE or COMPONENT:

- Once an ASSEMBLAGE or COMPONENT has ended, the values of any variables initialized within the ASSEMBLAGE or COMPONENT can be accessed through SOL's extended identifier notation.
- Chapter 7, section 7.2 details the scope rules governing variable initializations within ASSEMBLAGEs or COMPONENTs.

The extended identifier notation has the following syntax:

⟨ identifier ⟩ @ ⟨ path name ⟩

where:

⟨ identifier ⟩ is a legal SOL identifier

⟨ path name ⟩ specifies where the desired variable was initialized in the ASSEMBLAGE. This ⟨ path name ⟩ consists of either of the following:

1) an ASSEMBLAGE or COMPONENT name
2) an ASSEMBLAGE or COMPONENT name, followed by the symbol @ followed by another ⟨ path name ⟩

Extended identifiers describe a "search path," expressing the desired variable's location in the hierachy of an ASSEMBLAGE and its COMPONENTs. The ASSEMBLAGE is considered the OUTERMOST level, COMPONENTs directly inside the ASSEMBLAGE comprise the next most outer level, COMPONENTs directly inside COMPONENTs inside the ASSEMBLAGE are the third most outer level and so forth.

- The symbol @ means "of the." For example "weight@Flaps@Wing" should be read as, "the weight of the Flaps of the Wing." This reading makes clear which variable is being accessed.

- A search for the variable begins with the right-most name in the ⟨ path name ⟩ . The search differs slightly depending on where the extended identifier reference appears, as follows:

  i. The extended identifer reference appears inside an ASSEMBLAGE or COMPONENT:

     The search proceeds "outwards," (from the reference through the hierarchy towards the ASSEMBLAGE level). The search continues through the nested levels of COMPONENTs, checking the COMPONENT names at each level, until a matching COMPONENT or ASSEMBLAGE name is found. If the name is found, the search continues inwards from the matching COMPONENT or ASSEMBLAGE analagous with ii. (which follows), otherwise an error occurs.

     - A partial "path name" can be used, only the minimum length extended identifier that uniquely identifies a variable is required.

  ii. If the extended identifier reference appears outside the ASSEMBLAGE, the search moves from the ASSEMBLAGE inwards, matching the rightmost name in the "path name" with the ASSEMBLAGE name, the next rightmost name with a COMPONENT nested in the ASSEMBLAGE, the next name matches a COMPONENT nested in the COMPONENT in the ASSEMBLAGE and so on until the desired variable is found. If the variable is not found, an error occurs.

     - The full "path name" MUST be used in this case.

- SOL's extended identifier notation is best explained with the aid of the example which follows.

EXAMPLES:

Example 1: The following **ASSEMBLAGE** models an egg, and illustrates the use of SOL's extended identifier notation. (Some statements have been annotated to aid the discussion.)

```
PROGRAM Component_Demo

ASSEMBLAGE egg (0, ' ')
SUMMARIZE
  protein, weight
END SUMMARIZE
   protein_factor = .33

   COMPONENT Yolk (1, ' ')
      weight = 1.9
      protein = weight * protein_factor
   END Yolk

  print weight@Yolk@egg                                    ! 1)
  print weight@Yolk                                        ! 2)

   COMPONENT White (1, ' ')
      weight = 1.3
      protein = weight * protein_factor
   END White

  print weight@White@egg                                   ! 3)
  print weight@White                                       ! 4)
END egg

 print protein_factor@egg                                  ! 5)
 print weight@yolk@egg                                     ! 6)
 print weight@egg                                          ! 7)
END Component_Demo
```

The extended identifier notation is used in six print statements which write out the following values:

Print Statement	*Variable Accessed*	*REAL Number Value Printed*
1)	"weight of the Yolk of the egg"	1.9
2)	"weight of the Yolk"	1.9
3)	"weight of the White of the egg"	1.3
4)	"weight of the White"	1.3
5)	"protein_factor of the egg"	.33
6)	"weight of the yolk of the egg"	1.9
7)	"weight of the egg"	3.2

- Print statements 1) and *2)* print the same value. Recall that if an extended identifier reference appears INSIDE an ASSEMBLAGE or COMPONENT, only the minimum length extended identifier is required.
- Print statements 3) and 4) are analgous to statements 1) and 2).
- Notice that the complete "path name" must be given in statements 5), 6), and 7).
- The final print statement writes the value, 3.2. Because ASSEMBLAGE simple summarization variables are initialized with the sum of the appropriate simple summarization variables from its subCOMPONENTs. In this case, the "egg" COMPONENT has two subCOMPONENTs — "Yolk" and "White" – so that the "weight of the egg" equals "the weight of the Yolk" plus "the weight of the White," $1.9 + 1.3 = 3.2$.

The following restrictions apply to SOL's extended identifiers:

1) An extended identifier cannot appear on the left side of an assignment statement, or be passed as a subroutine dependent parameter. For example, the following is ILLEGAL:

```
weight@egg = 12
```

2) Extended identifiers can only be used to access variables that were initialized inside of an ASSEMBLAGE or COMPONENT.

3) Extended identifiers CANNOT be used to access variables before the variables have been initialized. (In particular, summarization variables of ASSEMBLAGEs or COMPONENTs which have nested sub COMPONENTs CANNOT be accessed until the summarization variables are automatically initialized when the ASSEMBLAGE or COMPONENT ends.)

7.2 SCOPE RULES FOR ASSEMBLAGEs AND COMPONENTs

ASSEMBLAGEs and COMPONENTs abide by the special scope rules. Because COMPONENT statements are often nested, it is useful to have terminology with which to distinguish the relationships among COMPONENTs to aid a discussion of scope rules. The following definitions are used:

Complete COMPONENT

— a COMPONENT whose body contains NO COMPONENT definition statements.

Composite COMPONENT

— COMPONENT whose body contains AT LEAST ONE COMPONENT definition statement.

Encompass

1) An ASSEMBLAGE ENCOMPASSes all COMPONENTs within it.

2) IF a Composite COMPONENT contains the COMPONENT definition statement for a second COMPONENT, the first COMPONENT is said to ENCOMPASS the second.

3) IF a Composite COMPONENT contains a sequence of nested COMPONENT definition statements, the composite COMPONENT is said to ENCOMPASS any and all members in the sequence.

Outer COMPONENT

— A term used to describe the relationship between COMPONENTs. ALL COMPONENTs that encompass a COMPONENT statement are OUTER to that COMPONENT. (The ASSEMBLAGE is by definition OUTER to all COMPONENTs).

Inner COMPONENT

— All COMPONENTs that a Composite COMPONENT encompasses are INNER to the Composite COMPONENT. (All COMPONENTs are by definition INNER to the ASSEMBLAGE).

SubCOMPONENT

— A term used to describe the relationship between COMPONENTs. All COMPONENTs whose definition statements appear IMMEDIATELY inside a COMPOSITE COMPONENT or ASSEMBLAGE are subCOMPONENTs of the composite COMPONENT or ASSEMBLAGE. For example:

```
COMPONENT one (1, ' ')
  COMPONENT Two (2, ' ')
    COMPONENT Three (3, ' ')
      sum_var = 2
    END Three
  END Two
  COMPONENT Four (2, ' ')
    sum_var = 6
  END Four
END one
```

- In this example, subCOMPONENTs Two and Four appear immediately inside COMPONENT one, but COMPONENT Three is NOT a subCOMPONENT of COMPONENT one COMPONENT Three is a subCOMPONENT of COMPONENT Two.

Other COMPONENT

— A term used to describe the relationship between COMPONENTs. All COMPONENTs which are neither an INNER nor an OUTER COMPONENT to a COMPONENT, are called OTHER components.

SCOPE RULES FOR ASSEMBLAGEs AND COMPONENTs:

Accessing or Altering Variables:

1) An ASSEMBLAGE or COMPONENT can access and alter variables which have been initialized in the MAIN program or SUBROUTINE implementation before the ASSEMBLAGE or COMPONENT statement appeared, For example:

```
c = 24
outer_var = 12
ASSEMBLAGE test (0, ' ')
SUMMARIZE
  summarize_var
END SUMMARIZE
  summarize_var = outer_var + 12/c ! access c and outer_var
  outer_var     = 42               ! alter outer_var
END test
```

- A warning message flags when an outer scope's variable is altered by an ASSEMBLAGE or COMPONENT such as the alteration to outer_var in this example.

2) An INNER COMPONENT can access and alter:

   i. Any variables initialized in the ASSEMBLAGE and OUTER COMPONENTs, if the variables are initialized BEFORE the inner COMPONENT appears.

   - A warning message flags when an outer scope's variable is altered.

3) An ASSEMBLAGE or COMPONENT can ACCESS but NOT ALTER variables initialized in INNER or OTHER COMPONENTs.

   i. Access using extended identifier notation. (See Chapter 7, section 7.1.2).

   ii. The INNER or OTHER COMPONENT variable must be initialized before it is accessed.

4) Variables initialized in an ASSEMBLAGE or COMPONENT can be ACCESSED but NOT ALTERED by main program or subroutine statements:

   i. Access with extended identifier notation. (See Chapter 7, section 7.1.2)

Initializing Variables:

1) Simple summarization variables and Expression summarization Variables are ALWAYS local. Every ASSEMBLAGE or COMPONENT accesses and initializes a local summarization variable.

2) When an ASSEMBLAGE or COMPONENT initializes a variable, a variable LOCAL to that ASSEMBLAGE or COMPONENT is initialized. This local variable can be accessed or altered outside the ASSEMBLAGE or COMPONENT ONLY by the rules above.

- **Note:** Variables are only initialized the first time they receive a value. Therefore, local variables are NOT created when an ASSEMBLAGE or COMPONENT assigns a value to an existing variable. The exceptions, of course, are summarization variables (see sections 7.1.1 and 7.1.1.1 of this chapter) because these variables are ALWAYS local.

EXAMPLES:

Example 1:

```
01 PROGRAM example
02
03   a = 6
04   ASSEMBLAGE pie (0, ' ')
05   SUMMARIZE
06     cost
07   END SUMMARIZE
08     cost = a + 1
09     a = 15
10     a = a + 2
11     print a
12   END pie
13   print cost@pie
14   print a
15 END example
```

What does the print statement on line 11 print?

- On line 9, a is given the value, 15 and then incremented by two on line 10. Therefore, on line 11 the value, 17 is printed.

What does the print statement on line 13 print?

- On line 8, the summarization variable cost is given the value a + 1. The variable a has the value 6 (from line 3) at that point, so cost is assigned the value 7, and 7 is printed.

What does the print statement on line 14 print?

- The assignment statements on lines 9 and 10 have altered the main program variable, a.. so the value 17 is printed.

Example 2:

```
01 PROGRAM example_2
02
03   factor = 13
04   ASSEMBLAGE one (0, ' ')
05   SUMMARIZE
06      a, b, cost = a * b
07   END SUMMARIZE
08
09     factor = 11
10     an_error = a@two@one
11     COMPONENT two (1, ' ')
12       a = factor + 2
13       b = a + factor
14       factor = 10
15     END two
16     COMPONENT three
17       a = factor + cost@two@one
18       b = b@two@one
19       factor = 3
20     END three
21     print factor
22   END one
23   PRINT a@two@one
24   PRINT b@three@one
25   PRINT a@three@one
26 END example_2
```

The rule numbers that appear in the discussion that follows refer to the rules for accessing and initializing variables that appear immediately before this "examples" section.

One error occurs in this program, Where is it?

- An error occurs on line 10. An **ASSEMBLAGE** can access an variables initialized in sub**COMPONENT**s, with extended identifier notation but only AFTER the variables have been initialized (as per rule 3 (ii) of Scope rules for accessing and Altering variables). **COMPONENT two** has not ended, so the summarization variables are not yet initialized and Line 10 is ILLEGAL. For the next questions, assume line 10 is removed so the program runs.

What value is printed on line 21?

- **factor** has been altered on lines 14 and 19 in accordance with rule 1, The assignment on line 19 is the most recent, therefore the value 3 is printed.

What value is printed on line 23?

- The **two COMPONENT**'s summarization variable a is assigned its value on line 12. The variable **factor** is referenced as per rule 1, and the value 11 is returned. Thus, **a@two** gets the value "11 + 2," and the value 13 is printed.

What value is printed on line 24?

- The **three** COMPONENT's summarization variable b is assigned its value on line 18 to have the same value as COMPONENT **two**'s summarization variable b. The value of the **two** COMPONENT's summarization variable b is set on line 13 to be "13 + 11" or 24. Thus, the value 24 is printed.

What value is printed on line 25 ?

- The **three** COMPONENT's summarization variable a is assigned its value on line 17. COMPONENT **two** has changed the value of **factor** to be "10" (line 14). COMPONENT **two**'s summarization variables a and b have the values "13" and "24" respectively as seen in the previous two print statements. Because the **cost** of COMPONENT **two** is an expression summarization variable, its value is computed automatically as a * b = 13 * 24 = 312. Thus, **factor** + 312 = 10 + 312 = 322, and the value 322 is printed.

7.3 ADVANCED MATERIAL — ASSEMBLAGE AND/OR COMPONENT ITERATION

The Iterations section of an **ASSEMBLAGE** or **COMPONENT** allows you to give initial values to ANY variables LOCAL to the **COMPONENT** or **ASSEMBLAGE**, such as summarization variables. The initial values are used before the variables get final values later in the **ASSEMBLAGE** or **COMPONENT** body.

Motivation: It is not always possible to create a sizing model where the sizing information can be handled in a linear fashion. The values for summarization variables of an **ASSEMBLAGE** are computed automatically to be the sum of the corresponding summarization variables of all nested subCOMPONENTs and are inherently linear. However, in some cases it would be helpful to define a subCOMPONENT's summarization variable in terms of another variable, such as an outer **COMPONENT** or **ASSEMBLAGE** summarization variable. For example, in modeling aircraft, the landing gear is often modeled as a fixed percentage of the total vehicle gross weight. The vehicle weight would be a summarization variable for the **ASSEMBLAGE**, and the weight of the landing gear would be a summarization variable of a subCOMPONENT.

Normally, such a model is not possible, because the total gross weight would not be initialized until the **ASSEMBLAGE** ended. The Iterations section allows proper modeling for these cases.

The iteration section for an **ASSEMBLAGE** is optional; when supplied the iteration section appears inside the **ASSEMBLAGE** summarization declaration section (As detailed in Chapter 7, section 7.1), and has the following syntax:

ITERATE
⟨ iterations ⟩

For example:

```
ASSEMBLAGE Test (0, ' ')
SUMMARIZE
  a, b
ITERATE
  a = 12
END SUMMARIZE
```

- The iteration section ends when the summarization declaration section ends with the words **END SUMMARIZE**.

The iteration section for a **COMPONENT** is optional; when supplied the iteration section appears immediately after the **COMPONENT** header (As detailed in Chapter 7, section 7.1), and has the following syntax:

```
ITERATE
    ⟨ iterations ⟩
END ITERATE
```

For example:

```
COMPONENT Test (0, ' ')
ITERATE
  a = 12
END ITERATE
```

where (for both **ASSEMBLAGE**s and **COMPONENT**s):

⟨ iterations ⟩ is one of the following:

1) ⟨ id ⟩ = ⟨ expr ⟩
2) ⟨ id ⟩ = ⟨ expr ⟩ : ⟨ no ⟩
3) ⟨ id ⟩ = ⟨ expr ⟩ : ⟨ no ⟩ %
4) MAX_ITERATIONS = ⟨ expr ⟩
5) a series of 1) and/or 2) and/or 3) and/or 4) separated by one or more carriage returns.

⟨ id ⟩ is a SOL identifier; the variable to be initialized.

⟨ expr ⟩ is an arithmetic expression for the initial value.

⟨ no ⟩ is a SOL number giving the convergence criteria.

The following restrictions apply to the ⟨ iterations ⟩ section, when it appears:

1) ⟨ id ⟩ must be a legal SOL identifier.
2) ⟨ id ⟩ CANNOT be a previously initialized variable, it MUST BE local to the **COMPONENT** which **iterates**. Since all variables initialized within a **COMPONENT** are local variables, any UNINITIALIZED variable can be used, including summarization variables.
3) ⟨ id ⟩ MUST BE INITIALIZED within the **COMPONENT** that declares the iteration. Initialization can be explicit (e.g. an assignment statement or subroutine call) or implicit (such as the automatic initialization of summarization variables of **COMPONENT**s as the sum of sub**COMPONENT** local summarization variables).
4) If no **Max_iterations** is explicitly given, a default value of fifty (50) is used. If iteration stops because **Max_iterations** is exceeded, a run-time error message will appear.

5) If no CONVERGENCE criteria is given, i.e. ": ⟨ *no* ⟩ " or ": ⟨ *no* ⟩ %," then a default absolute convergence criteria of 1 is used.

7.3.1 How Iteration Works

In what follows, all bracketed items refer to the iteration section syntax described in the previous section.

- The initial value for the variable is calculated in the ⟨ expr ⟩ part of an ⟨ iterations ⟩ section.
- The iteration variable can now appear in inner `COMPONENT` calculations.
- The iteration variable MUST be assigned (explicitly or implicitly) a new value in the `COMPONENT` or `ASSEMBLAGE` that defined the iteration.
- At the end of the `ASSEMBLAGE` or `COMPONENT` that initiated iteration, the new value of the iteration variable is compared to the value from the previous iteration.
- A *convergence criteria* defines whether the new/previous comparison will end the iteration or whether the new value will be used as the initial value and the process repeated. This repetition is called iteration.
- A convergence criteria is either ": ⟨ no ⟩ " (absolute convergence) or ": ⟨ no ⟩ % " (relative convergence). If nothing appears, a default absolute convergence criteria, with ⟨ no ⟩ = 1 is used.
- **absolute convergence**: When the difference between the previous value and the new value of the summarization variable is less than or equal to ⟨ no ⟩ , the iteration halts.
- **relative convergence**: When the previous value changes less than or equal to the percentage, " ⟨ no ⟩ %" then the iteration halts.
- `Max_iterations` places a further limit on the number of iterations allowed; the compare/repeat process will continue at most `max_iterations` times. If the convergence criteria is not satisfied at that point, iteration halts and an error message is output.

Example 1:

```
PROGRAM plane
!
! Demonstrates iteration
!
ASSEMBLAGE g_wt (0, 'all')
SUMMARIZE
     wt, vol, av_density = wt/vol
ITERATE
    wt = 10000:0.5 % ! iterate on wt
END SUMMARIZE

  COMPONENT wing (1, 'wing' )
     wt = 5000
     vol = 3000
  END wing

  COMPONENT Landing_gear (1, 'L.g' )
     wt = 0.03 * wt@g_wt
     vol = 30
  END Landing_gear

END g_wt
END plane
```

This example shows a landing gear modeling problem discussed earlier. Landing_gear is a subCOMPONENT of G_wt, but the value of its summarization variable, wt, is a function of the wt of the ASSEMBLAGE, G_wt, which is uninitialized until automatically initialized with the sum of its nested COMPONENTs. The initial value of wt@G_wt is given in the iteration section " wt = 10000 : 0.5 % ." The iteration statement can be read as, "use 10000 as the initial value for wt@G_wt in any calculation inside of G_wt." The calculation of wt@G_wt will repeat until it changes less than 0.5 % in consecutive iterations.

Chapter 8
The Optimize Statement

The **OPTIMIZE** statement acts as a sophisticated shell about the ADS[†] numerical optimization routine. Recall that one of SOL's purposes is to make the computer implementation of a numerical optimization problem as simple and error-free as possible. The **OPTIMIZE** statement permits the methods of numerical optimization implemented in the ADS optimization routine to be applied within a SOL program. The **OPTIMIZE** statement is combined with other SOL statements to pose an optimization problem; the resulting SOL program is compiled, linked and run to solve the posed problem and output the results.

This chapter discusses the **OPTIMIZE** statement syntax and use, with a minimal but necessary prior exposure to the concepts of numerical optimization assumed.

The **OPTIMIZE** statement has the following syntax:

```
OPTIMIZE ⟨ minimized variable ⟩
USE
    ⟨ design variables & constraints ⟩
    ⟨ Options Section ⟩
END USE
    ⟨ SOL statements ⟩
END OPTIMIZE
```

where:

⟨ minimized variable ⟩	is a legal SOL identifier, but CANNOT be an extended identifier.

[†] *ADS - A FORTRAN Program for Automated Design Synthesis — Version 1.10,* NASA Contractor Report 177985, Grant NAG1-567, 1985 by G.N. Vanderplaats

⟨ design variables & constraints ⟩ section is one of the following:

1) a design variable declaration
2) a constraint declaration
3) A series of design variable declarations or constraint declarations separated by one or more carriage returns.

- At least one design variable must appear in the `USE` section. The exact syntax for both 1) and 2) is given in sections 8.1 and 8.2 of this chapter.

⟨ Options Section ⟩ contains the optional settings for the optimizer, allows the user to specify a choice of optimization algorithms, to normalize design variables, to request output of optimization results, and to change default parameters of the ADS optimization routine. The optional settings are discussed in detail in section 8.3 of this chapter.

⟨ SOL statements ⟩ consists of one or more SOL statements. The objective function, ⟨ minimized variable ⟩ , and constraint functions are defined in this section.

The following restrictions apply to the `OPTIMIZE` statement:

1) The reserved word, `OPTIMIZE` and the ⟨ minimized variable ⟩ MUST appear alone on the same line. If they will not fit on a single line, the continuation symbol, &, must be used. For more information, see Chapter 2, section 2.4.

2) The reserved word, `USE`, must appear alone on a line.

3) The reserved words, `END USE`, must appear alone on the same line, and must be separated by at least one space (i.e. `ENDUSE` is illegal).

4) The reserved words, `END OPTIMIZE`, must appear alone on the same line, and must be separated by at least one space.

5) The ⟨ minimized variable ⟩ and the constraint variables (see section 8.2 which follows) MUST be initialized in the ⟨ SOL statements ⟩ section.

6) Design variables (see section 8.1 which follows) CANNOT be initialized or altered in the ⟨ SOL statements ⟩ section.

7) Any legal SOL statement (See Chapter 6) can appear in the ⟨ SOL statements ⟩ section.

   - Note that other `OPTIMIZE` statements can appear to perform *nested* optimizations.

8) The ⟨ minimized variable ⟩ MUST be a function of the design variables. This function is specified in the ⟨ SOL statements ⟩ section of the `OPTIMIZE` statement.

9) Values for the constraint variables MUST be specified in the ⟨ SOL statements ⟩ section.

The OPTIMIZE statement poses the optimization problem, "minimize the value of the variable, ⟨ minimized variable ⟩ by varying the values of the design variables and satisfying the constraints." The description of an optimization problem with an OPTIMIZE statement closely parallels the mathematical description of the problem, as seen in the following example:

Example:

Mathematical description:

Minimize: funct(x,y)
Subject to:

$-20 \leq x \leq 50$
$0 \leq y \leq 10$
$constraint(x, y) = 5$

Where:

$funct(x, y) = 10 * (y - x^2)^2 + (1 - x)^2$
$constraint(x, y) = x * y$

SOL program:

```
OPTIMIZE funct
USE

   x = -1.2 IN [-20, 50]
   y = 1 IN [0, 10]
   constraint .eq.  5

END USE

   funct = 10*(y - x**2)**2 + (1 - x)**2
   constraint = x * y
END OPTIMIZE
```

- The ⟨ minimized variable ⟩ is known as the *objective function,* and it is a function of the design variables. In the example above, the variable funct represents the objective function of two design variables, x and y.
- Design variables are stated in the ⟨ design variables & constraints ⟩ section.
  - By varying the values of the design variables, the value of the objective function is also varied.
  - Bounds on the values design variables can also be given to define the range of possible values. In the example, x and y are stated as bounded design variables.
  - The OPTIMIZE statement includes initial values for the design variables; the design variables of the example, x and y, are initialized to -1.2 and 1 respectively.
  - Section 8.2 of this Chapter describes design variables in greater detail.
- Constraints are stated in the ⟨ design variables & constraints section ⟩ . Constraints provide additional criteria which must be satisfied, beyond the goal of minimizing the objective function. For instance, we may want to minimize the weight of an airplane wing, but with the constraint that the wing cannot be too weak to withstand flight conditions. In the previous example, a single variable constraint represents the constraint function.
  - Like the objective function, constraints MUST be a function of the design variables.

- (ADVANCED MATERIAL) SOL constraint values are automatically *scaled* by the SOL compiler and constraint values are stored as a percentage of the constraint bound. If a particular constraint bound is zero, that constraint is left unscaled (See Section 8.2.1 of this chapter for details).

- The functions computing the ⟨ minimized variable ⟩ and the constraints MUST be stated in the ⟨ SOL statements ⟩ section.

The **OPTIMIZE** statement invokes the optimizer to solve the problem of minimizing the ⟨ minimized variable ⟩ The optimizer automatically varies the values of the design variables, increasing or decreasing the value of the ⟨ minimized variable ⟩ The optimizer finds the values for the design variables which minimize the value of the ⟨ minimized variable ⟩ and insures that all the constraints are satisfied. Some values for design variables may give a mimimum value to the ⟨ minimized variable ⟩ , but are ruled out because all the constraints cannot be satisfied with those values.

This Chapter is divided into the following sections:

8.1 DESIGN VARIABLE DECLARATIONS:

Design variable declarations, which appear in the ⟨ designs & constraints ⟩ section of an **OPTIMIZE** statement, have the following syntax:

⟨ design var ⟩ = ⟨ initial value ⟩ **IN** [⟨ lower bound ⟩ , ⟨ upper bound ⟩]

where:

⟨ design var ⟩ is a legal SOL identifier, and CANNOT be an extended identifier.

⟨ initial value ⟩ is an arithmetic expression. See Chapter 4, 4. 1 for more information on arithmetic expressions.

⟨ lower bound ⟩

⟨ upper bound ⟩ are either arithmetic expressions, or omitted. (they are optional; both may appear, only one may appear, or neither may appear).

- A comma MUST separate the upper and lower bounds even when one or both of the bounds is excluded.

The following restrictions apply to design variable declarations:

1) Design variable declarations MUST be separated from other design variable declarations or constraint declarations by one or more carriage returns.

2) Variables that appear in arithmetic expressions of ⟨ lower bound ⟩ , ⟨ initial value ⟩ , and ⟨ upper bound ⟩ CANNOT be design variables. For example:

Legal

```
extra_var = 1
OPTIMIZE mini
USE
  x = 1 IN [0, 2]
  y = 1 IN [extra_var, extra_var + 4]
END USE
```

Illegal

```
extra_var = 1
OPTIMIZE mini
USE
  x = 1 IN [0, 2]
  y = 1 IN [x, x + 4]
END USE
```

3) The entire design variable declaration must appear alone on a single line. If the declaration will not fit, then the continuation symbol & must be used. For further details, see Chapter 2, section 2.4.

4) The following relationships must be true:

- ⟨ lower bound ⟩ $\leq$ ⟨ initial value ⟩
- ⟨ initial value ⟩ $\leq$ ⟨ upper bound ⟩
- ⟨ lower bound ⟩ $<$ ⟨ upper bound ⟩

If the relationships above do not hold, a RUNTIME error message will be issued when your SOL program is executed. Such messages can only appear at runtime because arithmetic expressions whose values are not known at compile-time can be used for the bounds.

5) The ⟨ lower bound ⟩ and ⟨ upper bound ⟩ are calculated just once, in the design variable declaration. Subsequent changes to the variables used in the arithmetic expressions that compute the bounds will have no effect on the values of ⟨ lower bound ⟩ and ⟨ upper bound ⟩ .

6) It is illegal to alter or initialize a design variable inside an `OPTIMIZE` statement. (i.e. in the ⟨ SOL statements ⟩ part) The ADS optimizer controls the values of design variables.

7) Design variables must have unique names; two CANNOT have the same name nor can a design variable have the same name as a constraint.

8.2 CONSTRAINT DECLARATIONS:

The constraint declaration, which appears in the ⟨ designs & constraints ⟩ section of the `OPTIMIZE` statement, has the following syntax:

⟨ constraint var ⟩ ⟨ relationship ⟩ ⟨ bound ⟩

where:

⟨ constraint var ⟩ is a legal SOL identifier, but CANNOT be an extended identifier.

⟨ relationship ⟩ is one of the following:

.gt. means the value of ⟨ constraint_var ⟩ MUST be greater than the value of ⟨ bound ⟩

.lt. means the value of ⟨ constraint_var ⟩ MUST be less than the value of ⟨ bound ⟩

.eq. means the value of ⟨ constraint_var ⟩ MUST be equal to the value of ⟨ bound ⟩

⟨ bound ⟩ is an arithmetic expression. See Chapter 4, section 4.1 for more details on arithmetic expressions.

The following restrictions apply to the constraint declaration:

1) Constraint declarations MUST be separated from design variable declarations and other constraint declarations by one or more carriage returns.

2) Variables which appear in the ⟨ bound ⟩ expression CANNOT be design variables.

3) Variables which appear in the ⟨ bound ⟩ expression MUST be initialized BEFORE the constraint declaration.

4) The entire constraint declaration must appear alone on a single line. If the constraint declaration will not fit, the continuation symbol, &, must be used. For further details, see Chapter 2, section 2.4.

5) The ⟨ constraint var ⟩ MUST be initialized inside the ⟨ SOL statements ⟩ section of an `OPTIMIZE` statement, or a SOL error results.

6) A ⟨ constraint var ⟩ CANNOT be accessed before it is initialized.

7) Constraint variable names must be unique; two constraint variables CANNOT have the same name nor can a constraint variable and a design variable have the same name.

8.2.1 (ADVANCED MATERIAL) CONSTRAINT SCALING

SOL automatically scales constraint values as a percentage of the constraint limit (the ⟨ bound ⟩).

- The scaling is transparent to the user; only THE ACTUAL UNSCALED values are output to the user from a SOL program. Scaled values are used internally within the ADS optimizer.
- If a particular constraint limit is zero, then that particular constraint is left unscaled.

EXAMPLES:

The following example programs illustrate the OPTIMIZE statement and the declaration of design variables and constraints. The examples are deliberately simplistic so that syntactic and semantic details can be discussed without the complications of a a difficult optimization problem.

Example 1:

```
OPTIMIZE area
USE
  length = 5 IN [1, 10]
  width  = 5  IN [1, 10]
END USE
  area = length * width
END OPTIMIZE
PRINT 'the length is       :', length
PRINT 'the width is         :', width
PRINT 'the minimum area is :', area
```

This example is trivial, but illustrative. The problem posed is to minimize the value of the variable, "area." Two design variables are provided, "length" and "width;" no constraints are provided. The two design variables are bounded with a minimum value of one, and a maximum value of 10. The functional model states that, "area equals length times width." The optimizer will vary the values of "length" and "width" until the minimum area is found. In this case, the minimum area occurs when both the design variables are at minimum. Thus when the OPTIMIZE statement ends, the three print statements will print the following: "1," "1," and "1."

Example 2:

```
OPTIMIZE area
USE
  length = 5 IN [1, 10]
  width  = 5  IN [1, 5]
  diff .eq. 2.5
END USE
  area = length * width
  diff = length - width
END OPTIMIZE
PRINT 'the length is       :', length
PRINT 'the width is        :', width
PRINT 'the minimum area is :', area
```

This example differs from the previous example in only one respect: the addition of the constraint, "diff .eq. 2.5." The "diff" is the difference between the "length" and "width" as calculated by the statement, "diff = length - width."

The "diff" constraint effectively limits the range of possible values for "length." The ADS optimizer makes the values of "length" and "width" as small as possible, to minimize the "area." However, when "width" has a minimum value of "1," the "length" cannot be less than approximately "3.5" or the "diff" constraint would not be satisfied.

In this example, the optimizer varies the values of "length" and "width," minimizing the "area" while at the same time satisfying the "diff" constraint. When the OPTIMIZE statement ends, the three print statements will display the following: "1," "3.4902," and "3.4902."

Note: The "diff" constraint is only approximately satisfied; the difference between "length" and "width" is "2.5" in the exact solution. The variable, "diff," only approximates this exact value because the optimizer can only approximately satisfy constraints within a certain level of tolerance. There is a default tolerance, and the SOL programmer can adjust the tolerance through the ⟨ options ⟩ section. (See Chapter 8, section 8.3 for more information on ⟨ options ⟩)

The level of tolerance acts regardless of the difficulty of the problem. A very complex set of constraints from a difficult problem are satisfied with the same degree of precision as an easy problem. The optimizer approximately satisfied the "diff" constraint with the difference between the "length" and "width" equal "2.4902," very close to the exact value "2.5."

The OPTIMIZE statement may also be used to maximize a value, by minimizing the negative value. For example, the following simple OPTIMIZE statement maximizes the area of a rectangle:

Example 3:

```
OPTIMIZE neg_area
USE
  length = 5 IN [1, 10]
  width  = 5 IN [1, 10]
END
  neg_area = -(length * width)
END OPTIMIZE
area = -neg_area
```

This concludes the OPTIMIZE statement examples. A more tutorial presentation with examples appears in "The Sizing and Optimization Language, SOL — A Computer Language for Design Problems," NASA Technical Memorandum 100565, April 1988.

8.3 OPTIMIZE STATEMENT ⟨ OPTIONS SECTION ⟩ (ADVANCED MATERIAL)

The OPTIMIZE statement's ⟨ options section ⟩ allows the user to:

- The ⟨ options section ⟩ is optional; it need not appear in an OPTIMIZE statement.

When the ⟨ options section ⟩ appears, it has the following syntax:

OPTIONS
⟨ optional switches ⟩

where:

⟨ optional switches ⟩ is one or more of the following:

i. ⟨ strategy, optimizer, or search setting ⟩ This allows the user to select from a variety of optimization algorithms; a detailed description appears in section 8.3.1 of this chapter.

ii. ⟨ print results request ⟩ is a request to print the initial, intermediate or final results of an optimization; a detailed explanation appears in section 8.3.2 of this chapter.

iii. **normalize** This setting automatically normalizes the design variables, as described in section 8.3.3 of this chapter.

iv. ⟨ ADS parameter settings ⟩ These settings allow numerous values used by the ADS optimization routine, such as the constraint tolerance settings, to be altered to customize the optimization process. ADS provides default values, so that these control parameters need only be accessed when absolutely necessary The details are provided in section 8.3.4 of this chapter.

- Each of the ⟨ optional switches ⟩ MUST appear on a line alone, separated from other optional switches by one or more carriage returns.

For example, the following **OPTIMIZE** statement illustrates the use of the ⟨ options section ⟩ to select an optimizer, strategy combination, normalize design variables, and output some intermediate results (the design variable, constraint and SOL statement sections appear as comments):

Example:

```
OPTIMIZE energy
USE
   !
   !  Design variables and constraints would appear here
   !
OPTIONS
   optimizer = modified feasible directions
   strategy = sequential quadratic
   normalize
   print everything every iteration
END USE
  !
  !  equations would appear here
  !
END OPTIMIZE
```

The following rules apply to the ⟨ options section ⟩ :

1) The ⟨ optional switches ⟩ can appear in any order between the word OPTIONS and the end of the USE section (indicated by the words END USE).

2) Any number of switches can be used.

   - If a switch is repeated, the last value given is used by the optimizer.

8.3.1 *Strategy, optimizer and one-dimensional search settings (Advanced Material)*

The strategy, optimizer and one-dimensional search settings can appear in the ⟨ OPTIONS ⟩ section of an OPTIMIZE statement. These settings allow the selection of an optimization algorithm from those available within the ADS optimization software.

The strategy settings have the following syntax:

strategy = ⟨ strategy setting ⟩

where:

⟨ strategy setting ⟩ is one of the choices from table 8–1.

The optimizer settings have the following syntax:

optimizer = ⟨ optimizer setting ⟩

where:

⟨ optimizer setting ⟩ is one of the choices from table 8–2.

The one-dimensional search settings have the following syntax:

search = ⟨ search setting ⟩

where:

⟨ search setting ⟩ is one of the choices from table 8–3.

The tables that follow list the possible strategy, optimizer and one-dimensional search settings:

Table 8–1: Strategy settings:

Strategy Setting	*Strategy Used*
None	None, go directly to optimizer (*Default Setting*)
Exterior Penalty	Sequential unconstrained minimization using the exterior penalty function method.
Linear Penalty	Sequential unconstrained minimization using the linear extended interior penalty function method.
Quadratic Penalty	Sequential unconstrained minimization using the quadratic extended interior penalty function method.
Cubic Penalty	Sequential unconstrained minimization using the cubic extended interior penalty function method.
Lagrange Multiplier	Augmented Lagrange Multiplier method.
Sequential Linear	Sequential Linear Programming.
Inscribed Hyperspheres	Inscribed Hyperspheres (Method of Centers).
Sequential Quadratic	Sequential Quadratic Programming.
Sequential Convex	Sequential Convex Programming.

Table 8–2: Optimizer settings:

Optimizer Setting	*Optimization Method Used*
None	None. Go directly to the one-dimensional search. (This method should only be used for program development)
Fletcher-Reeves	Fletcher-Reeves algorithm for unconstrained minimization
DFP	Davidon-Fletcher-Powell (DFP) variable metric method for unconstrained minimization.
BFGS	Broyer-Fletcher-Goldfarb-Shanno (BFGS) method for unconstrained minimization. (*Default: unconstrained problems*)
Feasible Directions	Method of Feasible Directions (MFD) for constrained minimization.
Modified Feasible Directions	Modified Method of Feasible Directions for constrained minimization. (*Default: constrained problems*)

Table 8–3: One-dimensional search settings:

One-d Search Setting	*Search Method Used*
Golden Section	Find the minimum using the Golden Section Method.
Golden Section + Interpolation	Find the minimum using the Golden Section method followed by polynomial interpolation
Find Bounds + Interpolation	Find the minimum by first finding bounds and then using polynomial interpolation. (*Default Setting*)
Interpolation/Extrapolation	Find the minimum by polynomial interpolation/extrapolation without first finding bounds on the solution.

SOL automatically supplies default settings for search, optimizer, and one-dimensional search methods.

The default settings for constrained minimization are:

- `Strategy = None`
- `Optimizer = Modified Feasible Directions`
- `Search = Find Bounds + Interpolation`

The default settings for unconstrained minimization are:

- `Strategy = None`
- `Optimizer = BFGS` (Broyer-Fletcher-Goldfarb-Shanno).
- `Search = Find Bounds + Interpolation`

Recall that the ⟨ strategy, optimizer or search setting ⟩ appears within the ⟨ options section ⟩ of an `OPTIMIZE` statement. The following example illustrates the use of these settings:

```
OPTIMIZE weight
USE
  ! Design variables and constraints commented out
OPTIONS
  strategy  = Lagrange Multiplier
  optimizer = DFP
  search    = Golden Section
END USE
  ! optimization body commented out
END OPTIMIZE
```

As might be expected, not all combinations of strategy and optimizer settings are compatible. Table 8–4 (paraphrased from "ADS — A FORTRAN PROGRAM FOR AUTOMATED DESIGN SYNTHESIS — VERSION 1.10", NASA Contractor Report 177985, Grant NAG1–567, 1985 by G.N. Vanderplaats) identifies meaningful combinations of these two options:

Table 8–4: Meaningful Option Combinations:

Strategy	*Optimizer* Fletcher-Reeves	DFP	BFGS	FD	Modified FD
None	X	X	X	X	X
Exterior Penalty	X	X	X	0	0
Linear Penalty	X	X	X	0	0
Quadratic Penaly	X	X	X	0	0
Cubic Penalty	X	X	X	0	0
Lagrange Multiplier	X	X	X	0	0
Sequential Linear	0	0	0	X	X
Inscribed Hyperspheres	0	0	0	X	X
Sequential Quadratic	0	0	0	X	X
Sequential Convex	0	0	0	X	X

- The table pairs strategies with the optimizer selections.
- In this table, "X" denotes an acceptable strategy and optimizer combination.
- The appropriate (constrained or unconstrained) one-dimensional search is selected automatically.

8.3.2 Output of Optimization Results

SOL provides statements to request printing of the values of the objective function, design variables, constraints, and termination criteria at user-selected points during the optimization process. These ⟨ print results request ⟩ are placed in the `OPTIONS` section of an `OPTIMIZE` statement.

- The output produced by SOL ⟨ print results request ⟩ is integrated with SOL's `OPTIMIZE` statement.
- Although an output capability already exists within ADS, by default SOL suppresses the ADS output.
- The ADS output can be accessed via the `IPRINT` parameter setting, outlined in section 8.3.4 of this chapter.
- By default, the final values of the objective function, design variables, and all constraints are displayed at the termination of the optimization process.

The remainder of this section is divided in two parts:

I. discusses the syntax of the ⟨ print results request ⟩ that appears in the `OPTIONS` section.

II. discusses the format of the output display produced by a print results request.

I. Print Statements for Output of Optimization Results

A ⟨ print results requests ⟩ has the following syntax:

`PRINT` ⟨ optim value ⟩ ⟨ time ⟩

where:

⟨ optim value ⟩ is the optimization result to be printed, and can be one of the following:

`objective`	print the value of the objective function, the ⟨ minimized variable ⟩ .
`design variables`	print the values of the design variables.
`violated constraints`	print the values of violated constraints only.
`active constraints`	print the values of active and violated constraints only.
`constraints`	print the values of all contraints.
`termination criteria`	print values of the internal ADS optimization software variables used to terminate the optimization; primarily useful to the knowledgable user of ADS.
`everything`	print the current values of the objective function, design variables, constraints, and termination criteria.
`nothing`	negates ALL current print requests, including the default settings. The ⟨ time ⟩ parameter CANNOT appear with this setting.

⟨ time ⟩ specifies when to print during the optimization process.

- The ⟨ time ⟩ parameter is optional; it need not appear. When the ⟨ time ⟩ does not appear, the given ⟨ optim value ⟩ prints when the optimization ends (i.e. `at termination`).
- The ⟨ time ⟩ can be either a print time, or a series of print times separated by commas. If a series will not fit on a single line, the continuation symbol, &, must be used.
- A print time is one of the following:

	(nothing). When the ⟨ time ⟩ does not appear, the given ⟨ optim value ⟩ prints when the optimization ends (i.e. `at termination`).
`initially`	print the value(s) for ⟨ optim value ⟩ at the start of the optimization process.
`at termination`	print value(s) for ⟨ optim value ⟩ when optimization ends.
`every` ⟨ expr ⟩ `search step`	print the value(s) for ⟨ optim value ⟩ on the ⟨ expr ⟩ th iteration at the one-dimensional search level.

- The ⟨ expr ⟩ parameter is an optional SOL expression, just as in the case of optimizer/strategy iteration and is described in the next example).

every ⟨ expr ⟩ **iteration**

print the value(s) for ⟨ optim value ⟩ on the ⟨ expr ⟩ th iteration at either the strategy or optimizer level.

- If a strategy is used, printing occurs at the strategy level, otherwise printing occurs at the optimizer level.
- The ⟨ expr ⟩ parameter optional. If the ⟨ expr ⟩ parameter is not used, printing occurs on every iteration.
- When the ⟨ expr ⟩ parameter appears it is a SOL arithmetic expression (See Chapter 4, section 4.1.1).
- For example if no strategy is selected, `print objective every 2 iteration` specifies printing the value of the objective function at every 2^{nd} iteration of the optimizer.
- `print objective every 2/a iteration`, where `2/a = 4.7` at the time of the print request, specifies printing the value of the objective every 4^{th} iteration of the optimizer.
- The ⟨ expr ⟩ parameter is only evaluated once (before the iterative optimization process begins) and **REAL** values are truncated as in the last example above.

MULTIPLE PRINT REQUESTS:

- Print requests are USUALLY cumulative, with each print request being added to all previous print requests for that particular ⟨ optim value ⟩ .
- HOWEVER, some print requests SUPERCEDE previous print requests. For example:

```
PRINT violated constraints every iteration, at termination
PRINT constraints at termination
```

  will result in the violated constraints being printed **every iteration**, but all constraints will be printed **at termination** because of the second request, which supercedes the previous request that only violated constraints be output.
- When print requests contradict each other, the most recent request supercedes all others.

EXAMPLES:

The example which follows illustrates the use of print requests to output optimization results, with parts of the **OPTIMIZE** statement omitted and line numbers displayed for clarity:

Example:

```
0 : OPTIMIZE weight
1 : USE
2 :  ! Design variables and constraints would appear here
3 : OPTIONS
4 :   print objective initially, every iteration,
5 : &         every search step
6 :   print design variables every iteration
7 :   print everything at termination
8 : END USE
9 :   ! The rest of the optimization statement goes here
10: END OPTIMIZE
```

The print requests appear on lines 4–7.

- The print request that begins on line 4 illustrates the use of commas and a continuation symbol to specify a number of print times with a single print request.
- The ⟨ minimized variable ⟩ is **weight** (line 0), so that **weight** is the objective function and the print request on line 4 will display the value of **weight**.
- The print request on line 7 is nearly superfluous, as the final values of the objective, design variables and constraints are printed by default.

II. FORMAT OF OPTIMIZATION RESULTS OUTPUT

- The format of the output produced by SOL print requests is integrated with SOL language statements.
- The format consists of a title header followed by one or more of the following:
  - i. the value of the objective;
  - ii. the values for all design variables;
  - iii. the values for all or selected constraints;
  - iv. the values for the appropriate termination criteria variables.

Caveat: Values displayed at the strategy, optimizer, and one-dimensional search level are sometimes "incorrect." This is because the ADS optimization software uses scaled and/or normalized values at these levels. While the SOL output corrects for such scaling, the correction logic is immature and scaled and/or normalized values are occasionally displayed for the objective and constraints. The output format in no way affects the optimization process itself, and in any event the final values displayed are always correctly unscaled and denormalized. If this correction is not desired, the values internal to ADS are visible through the use of the ADS parameter IPRINT as discussed in section 8.3.4 of this chapter.

Each part of the format is discussed subsequently, and an example of the output format follows this discussion.

TITLE HEADER

The title header indicates the current point in the optimization process. The possible headers and associated points in the optimization process are as follows:

OPTIMIZATION INITIAL VALUES

Initial values at start of optimization process; **initially** setting.

OPTIMIZATION FINAL VALUES

Final values at the end of the optimization process; **at termination** setting.

OPTIMIZATION ONE-D SEARCH STEP ⟨ no ⟩

Values at the one dimensional search level; **every search step** setting.

OPTIMIZATION ITERATION NUMBER ⟨ no ⟩

Values at the optimizer level; **every iteration** setting. The actual iteration number is also displayed.

STRATEGY LEVEL ITERATION NUMBER ⟨ no ⟩

Values at the strategy level; **every iteration** setting when a strategy option has been selected. The actual iteration number is also displayed.

FINAL ITERATION NUMBER ⟨ no ⟩

This header is displayed at either the optimizer or strategy level just before termination; **every iteration** setting.

OBJECTIVE FUNCTION OUTPUT

The format for the objective function consists of the name of the SOL variable representing the objective, followed by an equals sign, = , followed by the current value of the objective. The value displayed is the current value being used by the ADS software.

Caveat: Objective function values from the strategy, optimizer, or one-dimensional search level are not always DISPLAYED "correctly," especially when equality constraints are used. The ADS software's internal representations of the objective are occasionally displayed, due to the immaturity of SOL's output formatting. This does not affect the optimization process and the final values are always correctly displayed.

DESIGN VARIABLES OUTPUT

The format for the design variables consists of the header **Design Variables Output** followed by a tabular listing. Each table row consists of the name of the design variable, its current value, and its bounds. In addition if a variable is at a bound, an exclamation point is displayed at the far right of the row. Unlike the objective and constraints, the "correct" values for the design variables are always displayed.

CONSTRAINTS OUTPUT

The format for the constraints consists of the header **Constraints Output** followed by a tabular listing. Each table row consists of the name of the constraint, its current value, the type of constraint (>, <, or =), its limit, and its current status (active, violated or satisfied).

Caveat: As with the objective function, values for the constraints are occassionally "incorrect," at the strategy, optimizer, or one-dimensional search level, as the ADS software's internal representations are sometimes displayed due to the immaturity of SOL's formatting. This does not affect the optimization process, and the final values are always correctly displayed.

TERMINATION CRITERIA OUTPUT

The termination criteria output simply displays the current values of the internal ADS variables and appropriate messages. This output option is primarily useful to the knowledgable user of ADS. The termination criteria format consists of the header **termination criteria**, followed by the appropriate values and messages. If none of the termination criteria are met, the header appears alone. The following are possible messages:

- **Maximum number of iterations exceeds/equals** followed by the number of optimizer or strategy iterations and the maximum allowable iterations.

- **Absolute convergence criteria is satisfied** followed by the current convergence criteria values.
- **Relative convergence criteria is satisfied** followed by the current convergence criteria values.
- **Kuhn Tucker Conditions are satisfied**
- **Kuhn Tucker Parameter** followed by the parameter value, **<=** , followed by the parameter limit.
- **Maximum K-T Residual** followed by the residual value, **<=** , followed by the limit.
- **Penalty exceeded limit** followed by the penalty, **>=** , followed by the limit.
- **Penalty below limit** followed by the penalty, **<=** , followed by the limit.
- **Kuhn Tucker Parameter** followed by the parameter value, **>=** , followed by the parameter limit.
- **S vector value** followed by the search vector value, **<=** , followed by the limit.

EXAMPLES:

The following is an example of optimization output produced by a print request. The title header indicates that the final results (at termination) are being displayed. Note the exclamation point at the far right of the design variable **inlet_pressure** indicating it has reached its bounds.

Example:

```
*********************************************************
* OPTIMIZATION FINAL RESULTS                            *
*********************************************************

OBJ_FLOWRATE                =     2.68346


                    *** DESIGN VARIABLES OUTPUT ***
                    *******************************

DESIGN VARIABLE              CURRENT VALUE                    BOUNDS

PANEL_FLOWRATE          =    2.6835        IN  [ 2.500     ,  10.00     ]
INLET_PRESSURE          =    4500          IN  [ 1200.     ,  4500.     ] !
PIN_H_O_D_PANEL1        =    1.0972        IN  [0.2000     ,  6.000     ]

                    *** CONSTRAINTS OUTPUT ***
                    **************************

CONSTRAINT NAME                VALUE        TYPE      LIMIT        STATUS

PINSTRESS_PANEL1             0.11436E-01     <       1.0000      SATISFIED
MICROWIDTH_PANEL1            0.54095E-02     >     0.50000E-02   ACTIVE
GAS_P_OUT                   -0.97474E+06     >       600.00      VIOLATED
```

8.3.3 Normalization of Design Variables

Design variables can be automatically normalized to the range $0 \ldots 1$ with the `normalize` directive, which can only appear in the `OPTIONS` section of an `OPTIMIZE` statement. The normalize directive has the following syntax:

```
normalize
```

The following equation was used for normalization. A design variable, `dv = x IN [ lower, upper ]`, is mapped to a new value, `dv = x' IN [0 , 1 ]` where:

$$x' = \frac{(x - lower)}{(upper - lower)}$$

For normalization to work properly, the following restrictions must be met:

1) The design variable's lower bound MUST be less than the upper bound.

2) The design variable's initial value, x, must be within the range:

$$lower \leq x \leq upper.$$

- Runtime error checking ensures the previous two conditions above are met.

3) Lower and upper bounds MUST be given or a SOL error results.

The ADS optimization software provides numerous parameters which can be set by the user to change the optimization process. These parameters can be set from within a SOL program, in the **OPTIONS** section of an **OPTIMIZE** statement. The parameter setting commands have the following syntax:

⟨ id ⟩ = ⟨ expr ⟩
where:

⟨ id ⟩ is a legal SOL identifier, and must be one of the ADS parameter names. (See table 8–5 that follows for parameter names).

⟨ expr ⟩ is an arithmetic expression. See Chapter 4, section 4.1.

- One or more of these ADS parameter setting commands can appear between the word **OPTIONS** and the end of the **OPTIMIZE USE** section.
- The ⟨ id ⟩ identifies the ADS parameter, and the ⟨ expr ⟩ identifies the new value for that parameter.

The following **OPTIMIZE** statement fragment illustrates the use of the **OPTIONS** section to change the values of the **ALAMDZ** and **IPRINT** ADS parameters:

Example:

```
OPTIMIZE min_var
USE
 x = 1 IN [2, 8]
OPTIONS
 ALAMDZ = 0.002
 IPRINT = 1111
END USE
```

All ADS parameter information given originates in "ADS — A FORTRAN PROGRAM FOR AUTOMATED DESIGN SYNTHESIS — VERSION 1.10", NASA Contractor Report 177985, Grant NAG1–567, 1985 by G.N. Vanderplaats.

The tables that follow list the possible ADS parameter names, the meaning of the parameters, and the default settings of the parameters

- The **ISTRAT, IOPT** and **IONED** parameters control the strategy, optimizer, and one dimensional search settings respectively (Discussed in section 8.3.1 of this chapter, tables 8–1, 8–2, and 8–3.) There should be no reason to use these three parameters since the SOL settings can be used to select strategy, optimizer, and one-dimensional search algorithms.

Table 8–5: Optimization Parameters:

ALAMDZ Initial estimate of the Lagrange Multipliers in the Augmented Lagrange Multiplier Method.

BETAMC Addditional steepest descent fraction in the method of centers. After moving to the center of the hypersphere, a steepest descent move is made equal to BETAMC times the radius of the hypersphere.

CTMIN Minimum constraint tolerance for nonlinear contraints. If a constraint is more positive than CTMIN, it is considered to be violated.

DABALP Absolute convergence criteria with one-dimensional search when using the Golden Section Method.

DABOBJ Maximum absolute change in the objective between two consecutive iterations to indicate convergence in optimization.

DABOBM Absolute convergence criterion for the optimization subproblem when using sequential minimization techniques.

DABSTR Same as DABOBJ, but used at the strategy level.

DELALP Relative convergence criteria for the one-dimensional search when using the Golden Section method.

DELOBJ Maximum relative change in the objective between two consecutive iterations to indicate convergence in optimization.

DELOBM Relative convergence criterion for the optimization subproblem when using sequential minimization techniques.

DELSTR Same as DELOBJ, but used at the strategy level.

DLOBJ1 Relative change in the objective function attempted on the first optimization iteration. Used to estimate initial move in the one-dimensional search. Updated as the optimization progresses.

DLOBJ2 Absolute change in the objective function attempted on the first optimization iteration. Used to estimate initial move in the one-dimensional search. Updated as the optimization progresses.

DX1 Maximum relative change in a design variable attempted on the first optimization iteration. Used to estimate the initial move in the one-dimensional search. Updated as the optimization progresses.

DX2 Maximum absolute change in a design variable attempted on the first optimization iteration. Used to estimate the initial move in the one-dimensional search. Updated as the optimization progresses.

EPSPEN Initial transition point for extended penalty function methods. Updated as the optimization progresses.

EXTRAP Maximum multiplier on the one-dimensional search parameter, ALPHA in the one-dimensional search using polynomial interpolation/extrapolation.

FDCH Relative finite difference step when calculating gradients.

FDCHM Minimum absolute value of the finite difference step when calculating gradients. This prevents too small a step when a design variable is near zero.

GMULTZ Initial penalty parameter in Sequential Quadratic Programming.

ICNDIR Restart parameter for conjugate direction and variable metric methods. Unconstrained minimization is restarted with a steepest descent direction every **ICNDIR** iterations.

IPRINT A four digit print control. **IPRINT =IJKL** where I,J,K and L have the following definitions.

I ADS system print control:
- 0 — No print.
- 1 — Print initial and final information
- 2 — Same as 1 plus parameter values and storage needs.
- 3 — Same as 2 plus scaling information calculated by ADS.

J Strategy print control.
- 0 — No print.
- 1 — Print initial and final optimization information.
- 2 — Same as 1 plus OBJ and X at each iteration.
- 3 — Same as 2 plus G at each iteration.
- 4 — Same as 3 plus intermediate information.
- 5 — Same as 4 plus gradients of contraints.

K Optimizer print control.
- 0 — No print.
- 1 — Print initial and final optimization information.
- 2 — Same as 1 plus OBJ and X at each iteration.
- 3 — Same as 2 plus constraints at each iteration.
- 4 — Same as 3 plus intermediate optimization and one-d search information.
- 5 — Same as 4 plus gradients of constraints.

L One-dimensional search print control.
- 0 — No Print.
- 1 — One-dimensional search debug information.
- 2 — More of the same.

ISCAL Scaling parameter. If ISCAL=0, no scaling is done. If ISCAL=1, the design variables, objective and constraints are scaled automatically.

ITMAX Maximum number of iterations allowed at the optimizer level.

ITRMOP The number of consecutive iterations for which the absolute or relative convergence criteria must be met to indicate convergence at the optimizer level.

ITRMST The number of consecutive iterations for which the absolute or relative convergence criteria must be met to indicate convergence at the strategy level.

JONED The one-dimensional search parameter (IONED) to be used in the Sequential Quadratic Programming method at the strategy level.

JTMAX Maximum number of iterations allowed at the strategy level.

PMULT Penalty multiplier for equality constraints when IOPT=4 or 5.

PSAIZ Move fraction to avoid constraint violations in Sequential Quadratic Programming.

RMULT Penalty function multiplier for the exterior penalty function method. Must be greater than 1.0.

RMVLMZ Initial relative move limit. Used to set the move limits in squential linear programming, method of inscribed hyperspheres and squential quadratic programming as a fraction of the value of a design variable.

RP	Initial penalty parameter for the exterior penalty function method or the Augmented Lagrange Multiplier method.
RPMAX	Maximum value of RP for the exterior penalty function method or the Augmented Lagrange Multiplier method.
RPMULT	Multiplier on RP for consecutive iterations.
RPPRIM	Initial penalty parameter for extended interior penalty function methods.
RRPMIN	Minimum value of RPPRIM to indicate convergence.
SCFO	The user-supplied value of the scale factor for the objective function if the default or calculated value is to be over-ridden.
SCLMIN	Minimum numerical value of any scale factor allowed.
STOL	Tolerance on the components of the caluculated search direction to indicate that the Kuhn-Tucker conditions are satisfied.
THETAZ	Nominal value of the push-off factor in the Method of Feasible Directions.
XMULT	Multiplier on the move parameter, ALPHA, in the one-dimensional search to find bounds on the solution.
ZRO	Numerical estimate of zero on the computer. Usually the default value is adequate.

The following table gives the default values for the switches listed above.

Table 8—6: Optimizer Switch default values.
f0 = the value of the objective with internal design variables equal zero.
NDV = the number of design variables.

Switch Name	*Default Value*
ALAMDZ	0.0
BETAMC	0.0
CT	−0.03 (If IOPT=4, CT =−0.1)
CTMIN	0.01
DABALP	0.0001 (If IONED=3 or 8, DABALP=0.001)
DABOBJ	ABS(f0)/1000
DABOBM	ABS(f0)/500
DABSTR	ABS(f0)/1000
DELALP	0.005 (If IONED=3 or 8, DELALP=0.05)
DELOBM	0.01
DELSTR	0.0001
DLOBJ1	0.1
DLOBJ2	1000.0
DX1	0.01
DX2	0.02
EPSPEN	−0.05
EXTRAP	5.0
FDCH	0.01
GMULTZ	10.0
ICNDIR	ndv+1
IPRINT	1000
ISCAL	1
ITMAX	40
ITRMOP	3
ITRMST	2
JONED	IONED
JTMAX	20
PSAIZ	0.95
PMULT	10.0
RMULT	5.0
RMVLMZ	0.2 (If ISTRAT=9, RMVLMZ=0.4)
RP	20.0
RPMAX	1.0E+10
RPMULT	0.2
RPPRIM	100.0
RRPMIN	1.0E−10
SCFO	1.0
SCLMIN	0.001
STOL	0.001
THETAZ	0.1
XMULT	2.618034
ZRO	0.00001

For further information about the use of these optimization switches, see "ADS—A FORTRAN PROGRAM FOR AUTOMATED DESIGN SYNTHESIS - VERSION 1.10", NASA Contractor report 177985, Grant NAG1-567, 1985 by G.N. Vanderplaats.

Chapter 9
Subroutines

Typically, a programming task can be decomposed into a number of simpler subproblems. Solutions to the subproblems are then combined to solve the main problem. In SOL, each subproblem can be coded as a subroutine.

A SOL subroutine consists of the following:

1) A subroutine name;

2) Zero or more *independent parameters* (input parameters);

   - Independent parameters specify variables which will be used, but NOT ALTERED by the subroutine.

3) Zero or more *dependent parameters* (output parameters);

   - Dependent parameters specify variables which will be altered or initialized by the subroutine.

4) optional declarations to be used within the subroutine;

5) A body of code that performs the subroutine's action;

- In this way, a subroutine associates a name, with a set of parameters and a body of statements.

To use a subroutine in a SOL program, do these three things:

1) Declare the subroutine in the main program declaration section. This step is detailed in Chapter 5, section 5.2, and in section 9.1.1 of this chapter.

2) Implement the subroutine in the subroutine implementation section of your SOL program. More details on subroutine implementation can be found in section 9.1.2 of this chapter.

3) Use a subroutine call statement to invoke the subroutine. A detailed discussion of the subroutine call statement can be found in Chapter 6, section 6.7, and in section 9.1.3 of this chapter.

This chapter is divided into three sections:

9.1 — Discusses subroutine declaration, implementation and calls.

9.2 — Discusses subroutine parameter passing conventions in detail.

9.3 — Discusses the scope rules that apply to subroutines.

9.1 SOL SUBROUTINES: DECLARATION, IMPLEMENTATION, AND CALLS

A SOL subroutine consists of three main elements:

1) Declaration:

   - The declaration that appears in the main program gives the subroutine name, and describes input/output behavior in terms of dependent and independent parameters. The SOL compiler can then check subroutine calls in the main program or subroutine implementations, and compare these calls against the corresponding declarations to insure the same number and type of parameters are supplied.

2) Call:

   - The subroutine call allows you to invoke a subroutine, and supply it with ACTUAL parameters. The subroutine returns values in the variables supplied as dependent parameters, by performing its action using the values of variables passed as independent variables. The SOL compiler insures that the correct number and type of variables are supplied as parameters. At runtime, if the call matches the declaration, the code provided in the subroutine's implementation will be executed. Once the subroutine call completes, the statement after the subroutine call is executed.

3) Implementation:

   - The subroutine implementation states the name of the subroutine, the dependent FORMAL parameters and the independent FORMAL parameters. The implementation MUST mirror the number, order, and types of parameters specified in the declaration section. The name of the subroutine must also be the same. The implementation supplies the code that will perform the subroutine's action. The dependent variables must be altered or initialized in this code, but error-checking insures that the independent parameters cannot be altered.

This section is further divided into the following subsections detailing the elements of subroutines:

9.1.1 SUBROUTINE DECLARATION

All SOL subroutines must be declared in the declaration section of the main program. A subroutine declaration has the following syntax:

SUBROUTINE (⟨ dependent_list ⟩) = ⟨ routine name ⟩ (⟨ independent list ⟩)

where:

⟨ dependent_list ⟩ is a parameter list consisting of one of the following:

1) Nothing, an empty list
2) A single identifier representing a parameter
   - Assumed to be of type REAL.
3) An identifier followed by a colon, and then a type name: REAL, INTEGER or LOGICAL.
4) A list of 2)'s or 3)'s or some combination of both, separated by commas.

⟨ routine name ⟩ is the name of the subroutine.

- It CANNOT be an extended identifier, see Chapter 2, section 2.4 or Chapter 7, section 7.1.2.
- You cannot use the subroutine name as the name of other SOL variables.

⟨ independent list ⟩ is syntactically the same as a ⟨ dependent_list ⟩

This syntax is also outlined in Chapter 5, section 5.2.

9.1.2 SUBROUTINE IMPLEMENTATION

A subroutine performs action. When you implement a subroutine, you do the following:

- Specify what action will be performed by the routine.
- Specify parameters will be used by the subroutine. In implementing a subroutine, only FORMAL parameters are specified.

Formal parameters are like variables in a formula; they are filled in later with specific values. When a subroutine is invoked by a subroutine call, the FORMAL input parameters are filled in with ACTUAL input parameters, and the subroutine's action is performed. This topic is discussed in greater detail in section 9.2 of this chapter. All SOL subroutines must be implemented in the subroutine implementation section that follows the main program.

A subroutine implementation has the following syntax:

SUBROUTINE (⟨ dependent list ⟩) = ⟨ routine name ⟩ (⟨ independent list ⟩)
⟨ optional declaration ⟩
⟨ subroutine body ⟩
END ⟨ name ⟩

where:

⟨ dependent list ⟩	is a parameter list, as outlined section 9.1.1 of this chapter, and represents the subroutine's formal dependent parameters.
⟨ routine name ⟩	is a legal SOL identifier, and is the name of the subroutine. • cannot be an extended identifier, see restriction 6) as follows.
⟨ independent list ⟩	is a parameter list, as outlined in section 9.1.1 of this chapter, and represents the subroutine's formal independent parameters.
⟨ optional declaration ⟩	is a SOL subroutine declaration section. The syntax for a declaration section is given in detail in chapter five. The subroutine declaration section shares an identical syntax with the main program declaration section with one exception: • no subroutines may be declared in the declaration section of a subroutine implementation.
⟨ subroutine body ⟩	is one or more SOL statements. Empty statements and blank lines can appear.

The following restrictions apply to subroutine implementations:

1) The subroutine ⟨ routine name ⟩ MUST be identical to the name declared in the main program declaration section.

2) The ⟨ dependent list ⟩ MUST be identical, in terms of type, number and order, to the dependent parameters declared in the main program declaration section, .

3) The ⟨ independent list ⟩ MUST be identical, in terms of type, number and order, to the independent parameters declared in the main program declaration section, .

4) A subroutine CANNOT alter the values of its independent parameters or a SOL error will result.

5) A subroutine MUST initialize its dependent parameters or a SOL error will result.

6) The subroutine ⟨ routine name ⟩ cannot be an extended identifier. See Chapter 2, section 2.4 or Chapter 7, section 7.1.2 for details on extended identifiers.

7) The names of all formal parameters must be unique.

8) A subroutine's body can only access variables initialized within the subroutine, or passed as parameters. Section 9.3 of this chapter explores scope rule restrictions in greater detail. Parameters are discussed in section 9.2 of this chapter.

When a subroutine is invoked by subroutine call:

- Formal independent parameters are initialized with actual parameter values before the subroutine's statements are executed.
- The statements in the subroutine implementation are executed.
- Formal dependent parameters are initialized by the subroutine statements.

When the execution of the subroutine statements is completed:

- the subroutine ends and the formal dependent parameters return their values to the actual dependent parameters.

9.1.3 The Subroutine Call

A subroutine is executed as a result of a subroutine call. The call consists of:

- the subroutine name
- the actual independent and dependent parameters.

The syntax for a subroutine call is:

(⟨ dependent parameters ⟩) = ⟨ routine name ⟩ (⟨ independent parameters ⟩)

For Example, the following are syntactically legal SOL subroutine calls:

1) `(a, b, c) = sub_one()`
2) `(a) = sub2(b, c)`
3) `() = empty_parameter_sub()`

- Chapter 6, section 6.7 offers a detailed discussion of the specific syntax of the subroutine call, detailing the syntax for the parameters and so on.

The following restrictions apply to subroutine calls:

1) If the subroutine declaration and subroutine implementation have parameters, the same number and type of parameters MUST be supplied when the routine is called.
2) If no parameters are specified in the declaration and implementation, then parameters CANNOT be supplied with the subroutine call.
3) SOL does not have recursion: SOL subroutine's should not call themselves. Furthermore, SOL subroutines should not call themselves indirectly, by calling another subroutine, or series of subroutines that eventually call the first routine.
    - The SOL compiler does NOT catch this error.

- The parameters supplied when calling a subroutine are referred to as ACTUAL parameters, to differentiate them from the FORMAL parameters specified by the subroutine declaration and implementation.
- The number and type of ACTUAL parameters MUST be the same as the number and type of FORMAL parameters (given in the subroutine declaration and implementation) or compile-time errors occur.

9.2 SUBROUTINE PARAMETERS

The parameters specified by the subroutine implementation are called the FORMAL parameters. The parameters given in a subroutine call are called the ACTUAL parameters. This distinction is explained in the three sections that follow:

9.2.1 - Discusses FORMAL parameters.

9.2.2 - Discusses ACTUAL parameters

9.2.3 - Discusses the association between FORMAL and ACTUAL parameters.

9.2.1 FORMAL PARAMETERS

A FORMAL parameter represents a local variable within the subroutine. These FORMAL parameters representing local variables can be accessed by the code that performs the subroutine's action, given in the subroutine implementation.

In this way, a FORMAL parameter acts like a variable in a formula. For example, $f(x) = 2x + 1$. The variable x is the FORMAL parameter; it designates a "blank" which is filled in later with an actual value. The actual values for FORMAL parameters are supplied when the subroutine is called.

There are two types of FORMAL parameters: independent and dependent, which are discussed in the two sections that follow.

9.2.1.1 Formal Independent Parameters

A FORMAL independent parameter represents a local variable within the subroutine implementation. An actual parameter is passed to the subroutine when it is called. The subroutine associates the VALUE of the actual parameter with the FORMAL independent parameter. The details on this association are supplied in section 9.2.3 of this chapter.

The following restrictions apply to INDEPENDENT parameters:

1) A subroutine cannot alter the value of a formal independent parameter. Thus, attempting to assign a value to a formal independent parameter is an error.

2) A subroutine CAN access the value of a formal independent parameter.

3) FORMAL INDEPENDENT parameters are INITIALIZED at the start of the subroutine implementation. (A very important fact, as in the following examples.)

4) The parameter names must be unique; no two parameters can have the same name.

Thus, the rule to remember about formal independent parameters is:

- "Use but do not Alter."

EXAMPLES:

The following sample subroutines illustrate legal and illegal usage of FORMAL independent parameters:

Example 1:

```
SUBROUTINE (a) = example_1(b, c)

! a is a formal dependent, b and c are formal
! independent parameters

  a = 12 * b + c
  c = 12
END example_1
```

- The line, `c = 12`, is ILLEGAL because it assigns a value to an formal independent parameter and a compilation error results.

Example 2:

```
SUBROUTINE (a) = example_2(b, c)

! a is a formal dependent, b and c are formal
! independent parameters

  a = 12 * b + c
  IF a .gt. 12 THEN
    c = 12
  END IF
END example_2
```

- The line, `c = 12`, is ILLEGAL. Recall that FORMAL independent parameters are initialized at the start of the subroutine implementation. Thus, `c` is initialized before the start of the IF statement, and the assignment within the `IF` statement is NOT to a local variable, `c`, but to the formal independent parameter. See Chapter 6, section 6.3.1 for details of IF statement scope rules.

Example 3:

```
SUBROUTINE (a) = example_3(b, c)

! a is a formal dependent, b and c are formal
! independent parameters

  a = 12 * b + c
END example_3
```

- This example is perfectly legal, as neither of the formal independent parameters have their values altered.

9.2.1.2 Formal Dependent Parameters

A formal dependent parameter represents a local variable within the subroutine. An actual parameter is associated with the formal dependent parameter when the subroutine is called. A value for the actual dependent parameter is returned by the formal dependent parameter when the subroutine ends. The details on this association are supplied in section 9.2.3 of this chapter.

The following restrictions apply to FORMAL DEPENDENT parameters:

1) A subroutine MUST initialize all FORMAL dependent parameters. Failing to assign a value to a FORMAL dependent parameter is an error.

2) A subroutine can access the value of a FORMAL dependent parameter, but only after it has been initialized within the subroutine.

3) The formal parameter names must be unique.

The subroutine directly accesses the actual parameter corresponding to the FORMAL dependent parameter. The FORMAL dependent parameters return the results of the subroutine's action; you MUST assign a new value to all FORMAL dependent parameters or a compile-time error will result.

EXAMPLES:

The following sample subroutines illustrate legal and illegal usage of formal dependent parameters.

Example 1:

```
SUBROUTINE (a, b) = example_1(c)

! a and b are formal dependent, c is a formal
! independent parameter

  a = 10.24 * c
  bad_try = a + b
END example_1
```

- The formal dependent parameter, b, is never initialized. This is an error.
- The line `bad_try = a + b` is ILLEGAL. Because b is not initialized, it cannot be used in an arithmetic expression. The SOL compiler issues a compile-time error message.

Example 2:

```
SUBROUTINE (a, b) = example_2(c)

! a and b are formal dependent, c is a formal
! independent parameter

  a = 12 + c
  IF a .gt. 12 THEN
    b = 12
  END IF
END example_2
```

- The formal dependent parameter, b, is never initialized. The line, b= 12 appears inside the THEN part of an IF statement, with no assignment to b in a corresponding ELSE part. Therefore, only a local variable, not the formal dependent parameter, is initialized. See Chapter 6, section 6.3.1 for more information on IF statement scope rules.

Example 3:

```
SUBROUTINE (a, b) = example_3(c)

! a and b are formal dependent, c is a formal
! independent parameter

  b = 6
  a = 12 * b + c
END example_3
```

This example is LEGAL. Both formal dependent parameters are initialized.

9.2.2 Actual Parameters

ACTUAL parameters are named when a subroutine is called. Earlier, it was stated that a FORMAL parameter was analagous to a variable in a formula, e.g., $f(x) = 2x + 1$.

- The variable x, is the FORMAL parameter which designates a "blank" to be filled in later.
- The actual value that fills in x is analgous to an actual parameter, e.g., f(2) and f(6) have 2 and 6 as actual independent parameters for x.
- The actual parameters for subroutines are given when the subroutine is called.

The following restrictions apply to ACTUAL parameters:

1) ACTUAL parameters MUST be variables; they CANNOT be subroutine calls, or literal values (e.g., .true. or 6) For example, in our $y = f(x)$ analogy, it is ILLEGAL to say f(2). Rather, we might say, q = 2, y = f(q).)

2) There is a one-to-one relationship between ACTUAL parameters and FORMAL parameters. The number and type of actual parameters MUST be identical to the number and type of FORMAL parameters or an error will result. The details of this relationship are discussed in section 9.2.3 of this chapter.

If an actual parameter is to be both accessed and altered in a subroutine, it can be passed as both an independent and dependent parameter. For example:

```
(x) = sub_1(x)
```

There are two types of actual parameters, dependent and independent, which are discussed in the sections that follow.

9.2.2.1 Actual Independent Parameters

An ACTUAL independent parameter is passed to a subroutine as input. For every ACTUAL independent parameter, there must be a corresponding FORMAL independent parameter. The associated FORMAL independent parameter takes the value of the ACTUAL independent parameter.

The value of actual INDEPENDENT parameters will not be altered by the subroutine. For example, the following demonstrates the use of actual independent parameters:

```
y = 2
(x) = sub_1(y)
print y
```

This subroutine will always print the real number "2". A subroutine cannot not alter its actual independent parameters; in this case we know what will be printed with no knowledge of the subroutine's action.

9.2.2.2 Actual Dependent Parameters

An ACTUAL dependent parameter is returned from a subroutine as output. For every ACTUAL dependent parameter, there must be a corresponding FORMAL dependent parameter. The associated formal dependent parameter corresponds directly to the actual dependent parameter. The value of the ACTUAL dependent parameter is assigned the value of the corresponding FORMAL dependent parameter when the subroutine ends.

EXAMPLES:

Assume that you have written an subroutine named square, with one dependent and one independent parameter. The subroutine returns the square of the independent parameter as the dependent parameter. A probable subroutine implementation follows:

```
SUBROUTINE (y) = square (x)
  y = x ** 2
END square
```

The following calls to subroutine square illustrate the principle of actual dependent parameters:

Example 1:

```
p = 3
(q) = square(p)
print q
```

This will print the real number 9, which equals 3^2

Example 2:

```
v = 5
(q) = square(v)
print v
print q
```

This will print the real number 5, which is v's value, and the number 25 which equals 5 * 5, q's value.

Thus, the value of the actual dependent parameter will be changed by the subroutine. Of course, it is possible for a subroutine to assign the actual parameter to its original value . Section 9.2.3 which follows provides a discussion on the association between actual and formal parameters.

9.2.3 The Relationship Between Actual and Formal Parameters

Actual parameters are named when a procedure is called, while formal parameters the names used by the subroutine implementation. Formal parameters are like "blanks" which are filled in by actual parameters provided at the call. SOL matches actual and formal parameters positionally (see example 2 which follows).

- Formal Independent parameters are *Copy-In*: The value of the actual parameter is copied into the formal independent parameter.
- Formal Dependent parameters are *by Reference*, the formal dependent parameter *references the same place in memory* as the actual parameter. Thus, any changes in the formal parameter within the subroutine affect the actual parameter. However, SOL assumes that dependent parameters are uninitialized, so that its error-checking assures that dependent parameters are assigned a value within the subroutine.

The best way to describe the relationship between actual and formal parameters is through an example. Consider the following subroutine implementation:

Example 1:

```
SUBROUTINE (a, b) = an_example (c, d)
   print c
   print d
   a = c * d
   b = c + d
END an_example
```

This subroutine has the formal dependent parameters, a and b, as well as formal independent parameters, c and d.

Example 2:

```
v = 4
w = 2
(y,z) = an_example (w , v)
print y
print z
```

In this call, y and z are named as actual dependent parameters, with w and v specified as actual independent parameters. Parameters are paired positionally:

— y is associated with a

— z is associated with b

— w is associated with c

— v is associated with d

The independent parameters are paired as follows:

— at the start of the subroutine, c = value of w, in other words c = 2.

— at the start of the subroutine, d = value of v, in other words, d = 4.

Wherever c and d are referenced in the subroutine the values 2 and 4, respectively, are used.
The dependent parameters are paired as follows:

— at the end of the subroutine, y = value of a, in other words y = 8. (8 = c * d = 2 * 4)

— at the end of the subroutine, z = value of b, in other words z = 6. (6 = c + d = 2 + 4)

Thus, when the print statements are reached, 8 and 6 are the values of y and z respectively.

9.3 THE SCOPE RULES FOR SUBROUTINES

In the body of a subroutine implementation, only three types of variables can be accessed:

1) The subroutine's independent formal parameters
   - The independent parameter's values can only be accessed, it is ILLEGAL to alter them.
2) The subroutine's dependent formal parameters can be accessed once they have been initialized within the subroutine.
3) Any variables initialized within the subroutine.

A subroutine CANNOT access any other variables, including but not limited to the following:

- variables initialized in the main program
- variables initialized in another subroutine

Chapter 10
Predeclared Functions

SOL provides predeclared functions that perform commonly used mathematical computations.

- Predeclared functions are invoked by a function reference.
- Predeclared functions use an actual argument as input, and return an arithmetic value.
- Functions can be referenced when evaluating arithmetic expressions or assignment statements. (See Chapter 4, section 4.1 for more information on arithemtic expressions).

SOL predeclared function references have the following syntax:

⟨ function name ⟩ (⟨ arith expr ⟩)

where:

⟨ function name ⟩ is the name of the predeclared function. Any other identifiers are illegal, and will result in an error.

⟨ arith expr ⟩ is a SOL arithmetic expression. See Chapter 4, section 4.1 for more information on arithmetic expressions.

SOL predeclared function references must abide by the following restrictions:

1) The ⟨ arith expr ⟩ must be of the proper type for the function referenced. The specific types needed by each function are detailed in the table that follows.

2) Functions can ONLY be referenced on the righthand side of an assignment statement, or, as part of an arithmetic expression.

3) Predeclared functions CANNOT be referenced as a subroutine parameter. (E.g., `subroutine_1(x, y, abs(x))`, where "abs" is a predeclared function, is ILLEGAL).

The following table details the purpose of each predeclared function, the number of arguments required, and the type of arguments required:

Table 10–1: Predeclared Functions

Purpose of Function	*Name*	*Number of Argument(s)*	*Type of Argument(s)*	*Type of Result(s)*
computes the absolute value of argument	ABS	1	REAL, INTEGER	REAL, INTEGER
computes the arc Tangent of argument	ATAN	1	REAL	REAL
computes the cosine of argument	COS	1	REAL	REAL
computes, e raised to the argument, $e^{argument}$	EXP	1	REAL	REAL
computes the natural logarithm of argument	LOG	1	REAL	REAL
truncates the value of the argument	INT	1	REAL	INTEGER
computes the sine of argument	SIN	1	REAL	REAL
computes the square root of argument	SQRT	1	REAL	REAL
computes the tangent of argument	TAN	1	REAL	REAL

EXAMPLES:

The following examples demonstrate function references in SOL:

Example 1:

```
R = 3.141 * ABS(-10.2 + 4)
```

R is assigned the value 3.141 multiplied by the absolute value of the sum of negative 10.2 and 4, (i.e. 6.2).

Example 2:

```
d = 12.00
p = cos(d)**2 + sin(d) ** 2
```

p is assigned the value of the cosine of twelve, squared, plus the value of the square of the sine of twelve. (i.e. $(\cos d)^2 + (\sin d)^2$)

Example 3:

```
d= EXP(2.0)
```

d is assigned the value of e to the second power, or e squared.

Example 4:

```
t = TAN(4)
```

This is illegal, because TANGENT requires a REAL argument, and "4" is an Integer. Note that "4.0" would have been legal. This error will NOT be caught by the SOL compiler, but will be discovered by the FORTRAN compiler, when the output of the SOL compiler is compiled.

Chapter 11
Scope Rules

SOL restricts how variables can be accessed to prevent uninitialized variables from being accessed. The *block* in which a variable is initialized determines the variable's *scope*, where in the program the variable can be accessed. The rules which determine where a variable can be accessed are called *scope rules*.

- Each of the following is a block in SOL:
  - i. the main program
  - ii. subroutines
  - iii. if/then/else statements
  - iv. assemblages and components
- The chief characteristic of a block is that a variable can be initialized inside a block, and remain uninitialized outside it.
  - For example, a variable can be initialized inside an `ASSEMBLAGE` but remain uninitialized in the SOL program outside the `ASSEMBLAGE` statement.
- Some blocks can be nested inside each other.
- A block is NOT the same as a multi-line statement. A multi-line statement simply extends over several lines. For example, the `OPTIMIZE` statement extends over several lines, but is NOT a block.

The following definitions are useful for describing SOL's variable access restrictions:

Block: A place in a SOL program where a variable can be initialized, and remain uninitialized in other parts of the program.

Scope: Refers to where (in a SOL program) a variable can be accessed. All the places where a variable can be accessed is the "scope" of the variable. The rules which decide where the variable can be accessed are called "scope rules."

Local: Used to describe the relationship between a variable and a block. Variables which are initialized inside a block are called "local" to that block.

This Chapter is divided into four sections:

11.1 MAIN PROGRAM SCOPE

1) The main program block has access to all identifiers initialized within it, and can call subroutines declared in the declaration section.
2) The main program can make variable type declarations in its declaration section.
3) The main program cannot call itself.
4) The main program block CANNOT access identifiers initialized inside of either:
   i. blocks that are nested inside the main program
   OR
   ii. subroutines.
   - **Exception:** variables initialized inside of `ASSEMBLAGE`s or `COMPONENT`s can be accessed with a special "extended identifier" notation. See Chapter 7, section 7.1.2
5) No variables can have the same name as an `ASSEMBLAGE`, `COMPONENT`, subroutine, the program name or other non-local variables.

11.2 SUBROUTINE SCOPE

1) A subroutine can access all identifiers initialized within it, which includes its own parameters.
2) A subroutine can make variable type declarations in its declaration section.
3) SOL subroutines can call any other subroutine declared in a given SOL program.
4) A subroutine cannot access the main program, or any identifiers initialized in the main program.
5) A subroutine cannot declare local subroutines
6) A subroutine cannot call itself. (*Recusion* is NOT permited.) The SOL compiler will not catch this error.

7) A subroutine CANNOT access identifiers initialized inside of either:

   i. blocks that are nested inside the subroutine.

   OR

   ii. other subroutines.

   - **Exception:** variables initialized inside of ASSEMBLAGEs or COMPONENTs within the subroutine can be accessed with a special "extended identifier" notation. See Chapter 7, section 7.1.2

- See Chapter 9 for further details on subroutines. Also, see Chapter 6, section 6.7 for information on subroutine calls.
- A brief overview of subroutines is presented in Chapter 2, section 2.1.4

11.3 IF/THEN/ELSE SCOPE

1) All previously initialized identifiers can be both accessed and altered within an IF/THEN/ELSE statement.

2) Assignments to a previously UNINITIALIZED variable within the THEN portion of an IF/THEN/ELSE statement will create and initialize a local variable, except under rule 4 which follows.

3) Assignments to a previously UNINITIALIZED variable within the ELSE portion of an IF/THEN/ELSE statement will create and initialize a local variable, except under rule 4 which follows.

4) An identifier which is initialized in both the THEN and the ELSE portions of an IF/THEN/ELSE statement is NOT local to the IF/THEN/ELSE statement. Instead, the variable is initialized in the block (i.e. the main program, a subroutine, an ASSEMBLAGE or COMPONENT, or even another IF/THEN/ELSE statement) that *encloses* the IF/THEN/ELSE.

- For instance, if an IF/THEN/ELSE statement appears in the main program, the main program *encloses* the IF/THEN/ELSE. But if an IF/THEN/ELSE statement appears INSIDE the THEN portion of a second IF/THEN/ELSE statement, the second IF/THEN/ELSE statement's THEN block *encloses* the first IF/THEN/ELSE statement.

5) Local variables of an IF/THEN/ELSE statement cannot be accessed outside the IF/THEN/ELSE block.

- Rules 2, 3, 4, and 5 were designed to insure that variables are not left uninitialized. Since variables are initialized the first time they get a value, a variable must be initialized in both the THEN and ELSE portion of IF/THEN/ELSE statements to be considered initialized in an outer block.
- Chapter 6, section 6.3 provides further details.

1) AT MOST ONE **ASSEMBLAGE** statement can appear within the main program.

2) AT MOST ONE **ASSEMBLAGE** statement can appear per subroutine implementation.

3) **COMPONENT** statements can ONLY appear within an **ASSEMBLAGE** statement or within another **COMPONENT** statement.

4) **ASSEMBLAGE**s and **COMPONENT**s cannot appear within **DO** loops.

5) **ASSEMBLAGE**s and **COMPONENT**s cannot appear within **IF/THEN/ELSE** statements.

6) Summarization variables and summarization expression variables are ALWAYS local variables.

   - **ASSEMBLAGE**s and **COMPONENT**s cannot initialize or alter another **ASSEMBLAGE** or **COMPONENT**'s summarization variables.
   - **ASSEMBLAGE**s and **COMPONENT**s can only access another **ASSEMBLAGE**s and **COMPONENT**'s summarization variables when: 1) extended identifier notation is used and 2) the **ASSEMBLAGE** or **COMPONENT** attempting access appears after the summarization variable is initialized.

7) **ASSEMBLAGE**s and **COMPONENT**s can ACCESS and ALTER all identifiers previously declared in outer blocks.

   - A warning message is issued when outer block variables are altered.

8) Any identifier initialized within an **ASSEMBLAGE** or **COMPONENT** cannot be altered by outer blocks.

   - However, the value of the variable can be accessed via the extended identifier notation.

- Chapter 7 provides a more detailed discussion of both **ASSEMBLAGE** scope rules and extended identifier notation.

Appendix A
BNF Grammar for SOL

What follows is a BNF (Backus-Naur Form) LALR(1) grammar for the Sizing and Opimization Language, SOL. In the grammar, the many semi-colon symbols represent carriage returns. In addition, ⟨ID ⟩ and ⟨NO ⟩ are terminal symbols representing SOL identifiers and numbers. This is the actual grammar used in the SOL compiler, so many of the grammar rules have been arranged to facilitate code emission.

⟨PROGRAM⟩ ::= ⟨?_;⟩⟨PROGRAM_MAIN⟩⟨PROGRAM_SUB?⟩⟨EOF⟩
⟨PROGRAM_SUB?⟩ ::=
⟨PROGRAM_SUB?⟩ ::= ⟨PROGRAM_SUB?⟩⟨PROGRAM_SUB⟩
⟨?_;⟩ ::=
⟨?_;⟩ ::= ⟨;⟩
⟨;⟩ ::= ;
⟨;⟩ ::= ⟨;⟩;
⟨PROGRAM_HEAD⟩ ::= ⟨PLAIN_PROGRAM_HEAD⟩⟨?_;⟩
⟨PLAIN_PROGRAM_HEAD⟩ ::= PROGRAM ⟨ID_PROGRAM⟩;
⟨PROGRAM_MAIN⟩ ::= ⟨PROGRAM_HEAD⟩⟨BLOCK⟩⟨STATEMENT_LIST⟩
+ ⟨END_ID⟩⟨;⟩
⟨PROGRAM_MAIN⟩ ::= ⟨PLAIN_PROGRAM_HEAD⟩⟨STATEMENT_LIST⟩
+ ⟨END_ID⟩⟨;⟩
⟨END_ID⟩ ::= END ⟨OK_ID⟩
⟨ID_PROGRAM⟩ ::= ⟨OK_ID⟩
⟨SUB_HEAD⟩ ::= ⟨SUBROUTINE⟩(⟨FORMAL_DEP_LIST?⟩) = ⟨ID_SUB_DEF⟩(
+ ⟨FORMAL_INDEP_LIST?⟩)
⟨PROGRAM_SUB⟩ ::= ⟨SUB_HEAD⟩⟨;⟩⟨BLOCK_SUB⟩
+ ⟨STATEMENT_LIST⟩⟨END_ID⟩⟨;⟩
⟨PROGRAM_SUB⟩ ::= ⟨SUB_HEAD⟩⟨STATEMENT_LIST⟩⟨END_ID⟩⟨;⟩
⟨SUBROUTINE⟩ ::= SUBROUTINE
⟨ID_SUB_DEF⟩ ::= ⟨OK_ID⟩
⟨FORMAL_DEP_LIST?⟩ ::=
⟨FORMAL_DEP_LIST?⟩ ::= ⟨FORMAL_DEP_LIST⟩
⟨FORMAL_DEP_LIST⟩ ::= ⟨X_ID⟩
⟨FORMAL_DEP_LIST⟩ ::= ⟨FORMAL_DEP_LIST⟩, ⟨X_ID⟩
⟨FORMAL_INDEP_LIST?⟩ ::=
⟨FORMAL_INDEP_LIST?⟩ ::= ⟨FORMAL_INDEP_LIST⟩
⟨FORMAL_INDEP_LIST⟩ ::= ⟨X_ID⟩
⟨FORMAL_INDEP_LIST⟩ ::= ⟨FORMAL_INDEP_LIST⟩, ⟨X_ID⟩
⟨X_ID⟩ ::= ⟨OK_ID⟩
⟨X_ID⟩ ::= ⟨OK_ID⟩: REAL
⟨X_ID⟩ ::= ⟨OK_ID⟩: INTEGER
⟨X_ID⟩ ::= ⟨OK_ID⟩: LOGICAL
⟨BLOCK⟩ ::= DECLARE ⟨MAIN_DECLARE⟩END DECLARE ;
⟨MAIN_DECLARE⟩ ::=
⟨MAIN_DECLARE⟩ ::= ⟨MAIN_DECLARE⟩⟨DECLARE_SUB⟩

```
⟨MAIN_DECLARE⟩          ::= ⟨MAIN_DECLARE⟩⟨DECLARE_TYPE⟩
⟨MAIN_DECLARE⟩          ::= ⟨MAIN_DECLARE⟩⟨FORTRAN_DECL⟩
⟨MAIN_DECLARE⟩          ::= ⟨MAIN_DECLARE⟩;
⟨DECLARE_SUB⟩           ::= ⟨SUBROUTINE⟩( ⟨DECLARE_DEP_LIST?⟩) = ⟨ID_SUB_DECL⟩(
+                           ⟨DECLARE_INDEP_LIST?⟩) ⟨;_DECLARE_SUB⟩
⟨DECLARE_DEP_LIST?⟩     ::=
⟨DECLARE_DEP_LIST?⟩     ::= ⟨DECLARE_DEP_LIST⟩
⟨DECLARE_DEP_LIST⟩      ::= ⟨W_ID⟩
⟨DECLARE_DEP_LIST⟩      ::= ⟨DECLARE_DEP_LIST⟩, ⟨W_ID⟩
⟨ID_SUB_DECL⟩           ::= ⟨OK_ID⟩
⟨DECLARE_INDEP_LIST?⟩   ::=
⟨DECLARE_INDEP_LIST?⟩   ::= ⟨DECLARE_INDEP_LIST⟩
⟨DECLARE_INDEP_LIST⟩    ::= ⟨W_ID⟩
⟨DECLARE_INDEP_LIST⟩    ::= ⟨DECLARE_INDEP_LIST⟩, ⟨W_ID⟩
⟨W_ID⟩                  ::= ⟨OK_ID⟩
⟨W_ID⟩                  ::= ⟨OK_ID⟩: REAL
⟨W_ID⟩                  ::= ⟨OK_ID⟩: INTEGER
⟨W_ID⟩                  ::= ⟨OK_ID⟩: LOGICAL
⟨;_DECLARE_SUB⟩         ::= ;
⟨BLOCK_SUB⟩             ::= DECLARE ⟨SUB_DECLARE⟩END DECLARE ;
⟨SUB_DECLARE⟩           ::=
⟨SUB_DECLARE⟩           ::= ⟨SUB_DECLARE⟩;
⟨SUB_DECLARE⟩           ::= ⟨SUB_DECLARE⟩⟨DECLARE_TYPE⟩
⟨SUB_DECLARE⟩           ::= ⟨SUB_DECLARE⟩⟨FORTRAN_DECL⟩
⟨TYPE_SPEC⟩             ::= REAL
⟨TYPE_SPEC⟩             ::= INTEGER
⟨TYPE_SPEC⟩             ::= LOGICAL
⟨DECLARE_TYPE⟩          ::= ⟨TYPE_SPEC⟩⟨TYPE_ID_LIST⟩;
⟨TYPE_ID_LIST⟩          ::= ⟨OK_ID⟩
⟨TYPE_ID_LIST⟩          ::= ⟨TYPE_ID_LIST⟩, ⟨OK_ID⟩
⟨FORTRAN_DECL⟩          ::= /* ;
⟨STATEMENT_LIST⟩        ::= ⟨;⟩
⟨STATEMENT_LIST⟩        ::= ⟨;⟩⟨STATEMENTS⟩
⟨STATEMENT_LIST⟩        ::= ⟨STATEMENTS⟩
⟨STATEMENTS⟩            ::= ⟨STATEMENT⟩
⟨STATEMENTS⟩            ::= ⟨STATEMENTS⟩;
⟨STATEMENTS⟩            ::= ⟨STATEMENTS⟩⟨STATEMENT⟩
⟨COMP_STMTS⟩            ::= ⟨STATEMENTS⟩
⟨STATEMENT⟩             ::= ⟨ASSEMBLAGE⟩
⟨STATEMENT⟩             ::= ⟨COMPONENT⟩
⟨STATEMENT⟩             ::= ⟨DO_STATEMENT⟩
⟨STATEMENT⟩             ::= ⟨IF_STATEMENT⟩
⟨STATEMENT⟩             ::= ⟨CALL_STATEMENT⟩
⟨STATEMENT⟩             ::= ⟨ASSIGNMENT⟩
⟨STATEMENT⟩             ::= ⟨OPTIM_STATEMENT⟩
⟨STATEMENT⟩             ::= ⟨PRINT_STATEMENT⟩
⟨STATEMENT⟩             ::= ⟨SUMMARIZE_STATEMENT⟩
⟨STATEMENT⟩             ::= /*
⟨PRINT_STATEMENT⟩       ::= ⟨PRINT⟩;
⟨PRINT_STATEMENT⟩       ::= ⟨PRINT⟩⟨PRINT_LIST⟩;
⟨PRINT⟩                 ::= PRINT
⟨OPT_SIGN⟩              ::=
⟨OPT_SIGN⟩              ::= +
⟨OPT_SIGN⟩              ::= -
⟨PRINT_TAIL⟩            ::= ⟨OPT_SIGN⟩⟨NO⟩: ⟨FORMAT⟩
⟨PRINT_TAIL⟩            ::= ⟨OPT_SIGN⟩⟨NO⟩
```

```
⟨PRINT_TAIL⟩                    ::= ⟨OK_ID⟩
⟨PRINT_TAIL⟩                    ::= ⟨OK_ID⟩: ⟨FORMAT⟩
⟨PRINT_TAIL⟩                    ::= ⟨STRING⟩
⟨PRINT_LIST⟩                    ::= ⟨PRINT_TAIL⟩
⟨PRINT_LIST⟩                    ::= ⟨PRINT_LIST⟩, ⟨PRINT_TAIL⟩
⟨FORMAT⟩                        ::= ⟨OK_ID⟩⟨NO⟩
⟨FORMAT⟩                        ::= ⟨OK_ID⟩
⟨SUMMARIZE_STATEMENT⟩           ::= ⟨SUMMARIZE_STATE_WORD⟩⟨SUMMARIZE_STATE_TAIL⟩;
⟨SUMMARIZE_STATE_WORD⟩          ::= SUMMARIZE
⟨SUMMARIZE_STATE_TAIL⟩          ::= ⟨OK_ID⟩
⟨SUMMARIZE_STATE_TAIL⟩          ::= ⟨SUMMARIZE_STATE_TAIL⟩, ⟨OK_ID⟩
⟨SUMMARIZE_STATE_TAIL⟩          ::= ⟨OK_ID⟩: ⟨FORMAT⟩
⟨SUMMARIZE_STATE_TAIL⟩          ::= ⟨SUMMARIZE_STATE_TAIL⟩, ⟨OK_ID⟩: ⟨FORMAT⟩
⟨DO_STATEMENT⟩                  ::= ⟨INDEXED_DO_STATEMENT⟩
⟨DO_STATEMENT⟩                  ::= ⟨LOGICAL_DO_STATEMENT⟩
⟨LOGICAL_DO_STATEMENT⟩          ::= ⟨LOGICAL_DO_HEAD⟩⟨LOGICAL_END_DO⟩;
⟨LOGICAL_DO_HEAD⟩               ::= DO ;
⟨LOGICAL_DO_HEAD⟩               ::= ⟨LOGICAL_DO_HEAD⟩⟨STATEMENT⟩
⟨LOGICAL_DO_HEAD⟩               ::= ⟨LOGICAL_DO_HEAD⟩;
⟨LOGICAL_END_DO⟩                ::= ⟨END_DO_WHEN⟩⟨LOGICAL_EXPRESSION⟩
⟨END_DO_WHEN⟩                   ::= ⟨END_DO⟩WHEN
⟨INDEXED_DO_STATEMENT⟩          ::= ⟨DO_HEAD⟩⟨END_DO⟩;
⟨END_DO⟩                        ::= END DO
⟨END_DO⟩                        ::= ENDDO
⟨DO_HEAD⟩                       ::= ⟨DO_BEGIN⟩, ⟨ARITH_EXPR⟩;
⟨DO_HEAD⟩                       ::= ⟨DO_HEAD⟩⟨STATEMENT⟩
⟨DO_HEAD⟩                       ::= ⟨DO_HEAD⟩;
⟨DO_BEGIN⟩                      ::= DO ⟨S_ID⟩= ⟨ARITH_EXPR⟩
⟨S_ID⟩                          ::= ⟨OK_ID⟩
⟨IF_STATEMENT⟩                  ::= ⟨IF_HEAD⟩⟨END_IF⟩;
⟨END_IF⟩                        ::= END IF
⟨END_IF⟩                        ::= ENDIF
⟨IF_STATEMENT⟩                  ::= ⟨IF_HEAD⟩⟨ELSE_HEAD⟩⟨END_IF⟩;
⟨IF_HEAD⟩                       ::= ⟨IF⟩⟨LOGICAL_EXPRESSION⟩THEN ;
⟨IF⟩                            ::= IF
⟨IF_HEAD⟩                       ::= ⟨IF_HEAD⟩⟨STATEMENT⟩
⟨IF_HEAD⟩                       ::= ⟨IF_HEAD⟩;
⟨ELSE_HEAD⟩                     ::= ELSE ;
⟨ELSE_HEAD⟩                     ::= ⟨ELSE_HEAD⟩⟨STATEMENT⟩
⟨ELSE_HEAD⟩                     ::= ⟨ELSE_HEAD⟩;
⟨ASSEMBLAGE⟩                    ::= ⟨ASSEM_HEADER⟩⟨STATEMENT_LIST⟩END ⟨OK_ID⟩;
⟨TAB_NO⟩                        ::= TAB ⟨NO⟩
⟨TAB_NO⟩                        ::= ⟨NO⟩
⟨ASSEM_HEADER⟩                  ::= ASSEMBLAGE ⟨ASSEM_ID⟩( ⟨OPT_SIGN⟩⟨TAB_NO⟩, ⟨STRING
+                                   ⟨;⟩⟨SUMMARIZE_DECLARE⟩
⟨ASSEM_ID⟩                      ::= ⟨OK_ID⟩
⟨SUMMARIZE_DECLARE⟩             ::= ⟨SUMMARIZE_WORD⟩⟨SUMMARIZE_TAIL⟩
+                                   ⟨ITERATIONS?⟩SUMMARIZE
⟨SUMMARIZE_WORD⟩                ::= SUMMARIZE ⟨;⟩
⟨SUMMARIZE_TAIL⟩                ::= ⟨SUMMARIZE_EXPR⟩⟨;⟩
⟨SUMMARIZE_EXPR⟩                ::= ⟨SUMMARIZE_STATE⟩
⟨SUMMARIZE_EXPR⟩                ::= ⟨SUMMARIZE_EXPR⟩, ⟨SUMMARIZE_STATE⟩
⟨SUMMARIZE_EXPR⟩                ::= ⟨SUMMARIZE_EXPR⟩⟨;⟩⟨SUMMARIZE_STATE⟩
⟨SUMMARIZE_STATE⟩               ::= ⟨SU_ID⟩
⟨SUMMARIZE_STATE⟩               ::= ⟨SUMMARIZE_ARITH_ASSIGN⟩
⟨SUMMARIZE_STATE⟩               ::= ⟨DECLARE_SWITCH⟩
```

```
⟨DECLARE_SWITCH⟩                ::= SUMMARY_TITLE = ⟨STRING⟩
⟨SU_ID⟩                         ::= ⟨OK_ID⟩
⟨SUMMARIZE_ARITH_ASSIGN⟩        ::= ⟨SU_AR_ID⟩⟨S_=⟩⟨S_ARITH_EXPR⟩
⟨SU_AR_ID⟩                      ::= ⟨OK_ID⟩
⟨S_=⟩                           ::= =
⟨S_ARITH_EXPR⟩                  ::= ⟨S_TERM⟩
⟨S_ARITH_EXPR⟩                  ::= ⟨S_ARITH_EXPR⟩⟨S_+⟩⟨S_TERM⟩
⟨S_ARITH_EXPR⟩                  ::= ⟨S_ARITH_EXPR⟩⟨S_-⟩⟨S_TERM⟩
⟨S_ARITH_EXPR⟩                  ::= ⟨S_+⟩⟨S_TERM⟩
⟨S_ARITH_EXPR⟩                  ::= ⟨S_-⟩⟨S_TERM⟩
⟨S_+⟩                           ::= +
⟨S_-⟩                           ::= -
⟨S_TERM⟩                        ::= ⟨S_PRIMARY⟩
⟨S_TERM⟩                        ::= ⟨S_TERM*⟩⟨S_PRIMARY⟩
⟨S_TERM*⟩                       ::= ⟨S_TERM⟩*
⟨S_TERM⟩                        ::= ⟨S_TERM/⟩⟨S_PRIMARY⟩
⟨S_TERM/⟩                       ::= ⟨S_TERM⟩/
⟨S_PRIMARY⟩                     ::= ⟨S_SOURCE⟩
⟨S_PRIMARY⟩                     ::= ⟨S_PRIMARY_HEAD⟩⟨S_SOURCE⟩
⟨S_PRIMARY_HEAD⟩                ::= ⟨S_PRIMARY⟩**
⟨S_SOURCE⟩                      ::= ⟨NO⟩
⟨S_SOURCE⟩                      ::= ⟨OK_ID⟩
⟨S_SOURCE⟩                      ::= ⟨S_(_EXPRESSION⟩⟨S_ARITH_EXPR⟩)
⟨S_(_EXPRESSION⟩                ::= (
⟨S_SOURCE⟩                      ::= ⟨S_FUNCT⟩⟨S_ARITH_EXPR⟩)
⟨S_FUNCT⟩                       ::= SIN (
⟨S_FUNCT⟩                       ::= ABS (
⟨S_FUNCT⟩                       ::= SQRT (
⟨S_FUNCT⟩                       ::= COS (
⟨S_FUNCT⟩                       ::= LOG (
⟨S_FUNCT⟩                       ::= TAN (
⟨S_FUNCT⟩                       ::= ATAN (
⟨S_FUNCT⟩                       ::= INT (
⟨S_FUNCT⟩                       ::= EXP (
⟨COMPONENT⟩                     ::= ⟨COMP_WORD⟩⟨ID_COMP⟩( ⟨OPT_SIGN⟩⟨TAB_NO⟩, ⟨STRING
+                                   ⟨COMP_ITERATIONS?⟩⟨COMP_STMTS⟩END ⟨OK_ID⟩;
⟨COMP_WORD⟩                     ::= COMPONENT
⟨COMP_WORD⟩                     ::= COMP
⟨ID_COMP⟩                       ::= ⟨OK_ID⟩
⟨COMP_ITERATIONS?⟩              ::=
⟨COMP_ITERATIONS?⟩              ::= ⟨ITERATING⟩ITERATE ⟨;⟩
⟨ITERATIONS?⟩                   ::= END
⟨ITERATIONS?⟩                   ::= ⟨ITERATING⟩
⟨ITERATING⟩                     ::= ITERATE ; ⟨COMP_SWITCHES⟩; END
⟨COMP_SWITCHES⟩                 ::= ⟨COMP_SW_ID_1⟩= ⟨ARITH_EXPR⟩⟨CONVERGE?⟩
⟨COMP_SWITCHES⟩                 ::= ⟨COMP_SWITCHES⟩⟨;⟩⟨COMP_SW_ID⟩= ⟨ARITH_EXPR⟩
+                                   ⟨CONVERGE?⟩
⟨CONVERGE?⟩                     ::=
⟨CONVERGE?⟩                     ::= : ⟨NO⟩
⟨CONVERGE?⟩                     ::= : ⟨NO⟩%
⟨COMP_SW_ID_1⟩                  ::= ⟨OK_ID⟩
⟨COMP_SW_ID⟩                    ::= ⟨OK_ID⟩
⟨CALL_STATEMENT⟩                ::= (⟨_CALL⟩⟨ACTUAL_DEP_LIST?⟩) = ⟨ID_SUB_CALL⟩
+                                   ( ⟨ACTUAL_INDEP_LIST?⟩) ;
⟨(_CALL⟩                        ::= (
⟨ID_SUB_CALL⟩                   ::= ⟨OK_ID⟩
```

⟨ACTUAL_DEP_LIST?⟩ ::=
⟨ACTUAL_DEP_LIST?⟩ ::= ⟨ACTUAL_DEP_LIST⟩
⟨ACTUAL_DEP_LIST⟩ ::= ⟨Y_ID⟩
⟨ACTUAL_DEP_LIST⟩ ::= ⟨ACTUAL_DEP_LIST⟩, ⟨Y_ID⟩
⟨ACTUAL_INDEP_LIST?⟩ ::=
⟨ACTUAL_INDEP_LIST?⟩ ::= ⟨ACTUAL_INDEP_LIST⟩
⟨ACTUAL_INDEP_LIST⟩ ::= ⟨Y_ID⟩
⟨ACTUAL_INDEP_LIST⟩ ::= ⟨ACTUAL_INDEP_LIST⟩, ⟨Y_ID⟩
⟨Y_ID⟩ ::= ⟨OK_ID⟩
⟨ASSIGNMENT⟩ ::= ⟨Z_ID⟩⟨=⟩⟨ARITH_EXPR⟩;
⟨ASSIGNMENT⟩ ::= ⟨Z_ID⟩⟨=⟩⟨LOGICAL_EXPRESSION⟩;
⟨Z_ID⟩ ::= ⟨OK_ID⟩
⟨=⟩ ::= =
⟨OPTIM_STATEMENT⟩ ::= ⟨OPTIM_HEAD⟩⟨;⟩⟨OPTIM_BODY⟩END OPTIMIZE ;
⟨OPTIM_HEAD⟩ ::= OPTIMIZE ⟨OK_ID⟩
⟨OPTIM_BODY⟩ ::= ⟨USE_STATEMENT⟩
⟨OPTIM_BODY⟩ ::= ⟨OPTIM_BODY⟩⟨STATEMENT⟩
⟨OPTIM_BODY⟩ ::= ⟨OPTIM_BODY⟩;
⟨USE_HEAD⟩ ::= USE ⟨;⟩
⟨USE_STATEMENT⟩ ::= ⟨USE_HEAD⟩⟨USE_BODY⟩⟨OPTIONS?⟩END USE ;
⟨OPTIONS?⟩ ::=
⟨OPTIONS?⟩ ::= ⟨OPTIONS_HEAD⟩⟨O_LIST⟩
⟨OPTIONS_HEAD⟩ ::= OPTIONS ⟨;⟩
⟨O_LIST⟩ ::= ⟨OPTIM_PRINT⟩
⟨O_LIST⟩ ::= ⟨ALGO_STMT⟩
⟨O_LIST⟩ ::= ⟨OPTIM_SWITCHES⟩⟨;⟩
⟨O_LIST⟩ ::= ⟨O_LIST⟩⟨OPTIM_PRINT⟩
⟨O_LIST⟩ ::= ⟨O_LIST⟩⟨ALGO_STMT⟩
⟨O_LIST⟩ ::= ⟨O_LIST⟩⟨OPTIM_SWITCHES⟩⟨;⟩
⟨ALGO_STMT⟩ ::= STRATEGY = ⟨STRATEGY⟩⟨;⟩
⟨ALGO_STMT⟩ ::= OPTIMIZER = ⟨OPTIMIZER⟩⟨;⟩
⟨ALGO_STMT⟩ ::= SEARCH = ⟨SEARCH⟩⟨;⟩
⟨STRATEGY⟩ ::= NONE
⟨STRATEGY⟩ ::= EXTERIOR PENALTY
⟨STRATEGY⟩ ::= LINEAR PENALTY
⟨STRATEGY⟩ ::= QUADRATIC PENALTY
⟨STRATEGY⟩ ::= CUBIC PENALTY
⟨STRATEGY⟩ ::= LAGRANGE MULTIPLIER
⟨STRATEGY⟩ ::= SEQUENTIAL LINEAR
⟨STRATEGY⟩ ::= INSCRIBED HYPERSPHERES
⟨STRATEGY⟩ ::= SEQUENTIAL QUADRATIC
⟨STRATEGY⟩ ::= SEQUENTIAL CONVEX
⟨OPTIMIZER⟩ ::= NONE
⟨OPTIMIZER⟩ ::= FLETCHER - REEVES
⟨OPTIMIZER⟩ ::= DFP
⟨OPTIMIZER⟩ ::= BFGS
⟨OPTIMIZER⟩ ::= FEASIBLE DIRECTIONS
⟨OPTIMIZER⟩ ::= MODIFIED FEASIBLE DIRECTIONS
⟨SEARCH⟩ ::= GOLDEN SECTION
⟨SEARCH⟩ ::= GOLDEN SECTION + INTERPOLATION
⟨SEARCH⟩ ::= FIND BOUNDS + INTERPOLATION
⟨SEARCH⟩ ::= INTERPOLATION/EXTRAPOLATION
⟨OPTIM_PRINT⟩ ::= PRINT NOTHING ⟨;⟩
⟨OPTIM_PRINT⟩ ::= PRINT ⟨OPTIM_VAL⟩⟨OPT_PRT_TIME?⟩⟨;⟩
⟨OPTIM_VAL⟩ ::= OBJECTIVE
⟨OPTIM_VAL⟩ ::= DESIGN VARIABLES

⟨OPTIM_VAL⟩ ::= VIOLATED CONSTRAINTS
⟨OPTIM_VAL⟩ ::= ACTIVE CONSTRAINTS
⟨OPTIM_VAL⟩ ::= CONSTRAINTS
⟨OPTIM_VAL⟩ ::= TERMINATION CRITERIA
⟨OPTIM_VAL⟩ ::= EVERYTHING
⟨OPT_PRT_TIME?⟩ ::=
⟨OPT_PRT_TIME?⟩ ::= ⟨PRT_TIME⟩
⟨PRT_TIME⟩ ::= ⟨TIME⟩
⟨PRT_TIME⟩ ::= ⟨PRT_TIME⟩, ⟨TIME⟩
⟨TIME⟩ ::= INITIALLY
⟨TIME⟩ ::= EVERY ⟨?OPT_EXPR⟩ITERATION
⟨TIME⟩ ::= EVERY ⟨?OPT_EXPR⟩SEARCH STEP
⟨TIME⟩ ::= AT TERMINATION
⟨?OPT_EXPR⟩ ::=
⟨?OPT_EXPR⟩ ::= ⟨ARITH_EXPR⟩
⟨OPTIM_SWITCHES⟩ ::= ⟨OPTIM_SET⟩
⟨OPTIM_SWITCHES⟩ ::= ⟨OPTIM_SWITCHES⟩⟨OPTIM_SET⟩
⟨OPTIM_SET⟩ ::= NORMALIZE
⟨OPTIM_SET⟩ ::= ⟨OPTIM_SW_ID⟩= ⟨ARITH_EXPR⟩
⟨OPTIM_SW_ID⟩ ::= ⟨OK_ID⟩
⟨USE_BODY⟩ ::= ⟨USE_DESIGN_OR_CONSTRAINT⟩
⟨USE_BODY⟩ ::= ⟨USE_BODY⟩⟨USE_DESIGN_OR_CONSTRAINT⟩
⟨USE_DESIGN_OR_CONSTRAINT⟩ ::= ⟨D_ID⟩= ⟨DES⟩IN [⟨DES_LO⟩, ⟨DES_HI⟩] ⟨;⟩
⟨D_ID⟩ ::= ⟨OK_ID⟩
⟨DES_LO⟩ ::=
⟨DES_LO⟩ ::= ⟨ARITH_EXPR⟩
⟨DES⟩ ::= ⟨ARITH_EXPR⟩
⟨DES_HI⟩ ::=
⟨DES_HI⟩ ::= ⟨ARITH_EXPR⟩
⟨USE_DESIGN_OR_CONSTRAINT⟩ ::= ⟨C_ID⟩⟨FUZZY_RELAT⟩⟨ARITH_EXPR⟩⟨;⟩
⟨C_ID⟩ ::= ⟨OK_ID⟩
⟨FUZZY_RELAT⟩ ::= .EQ.
⟨FUZZY_RELAT⟩ ::= .LT.
⟨FUZZY_RELAT⟩ ::= .GT.
⟨LOGICAL_EXPRESSION⟩ ::= ⟨LOGICAL_FACTOR⟩
⟨LOGICAL_EXPRESSION⟩ ::= ⟨LOGICAL_EXPRESSION⟩⟨.OR.⟩⟨LOGICAL_FACTOR⟩
⟨.OR.⟩ ::= .OR.
⟨LOGICAL_FACTOR⟩ ::= ⟨LOGICAL_SECONDARY⟩
⟨LOGICAL_FACTOR⟩ ::= ⟨LOGICAL_FACTOR⟩⟨.AND.⟩⟨LOGICAL_SECONDARY⟩
⟨.AND.⟩ ::= .AND.
⟨LOGICAL_SECONDARY⟩ ::= ⟨LOGICAL_PRIMARY⟩
⟨LOGICAL_SECONDARY⟩ ::= ⟨.NOT.⟩⟨LOGICAL_PRIMARY⟩
⟨.NOT.⟩ ::= .NOT.
⟨LOGICAL_PRIMARY⟩ ::= ⟨ARITH_EXPR⟩⟨RELAT⟩⟨ARITH_EXPR⟩
⟨LOGICAL_PRIMARY⟩ ::= .TRUE.
⟨LOGICAL_PRIMARY⟩ ::= .FALSE.
⟨LOGICAL_PRIMARY⟩ ::= ⟨OK_ID⟩
⟨(_EXPRESSION⟩ ::= (
⟨LOGICAL_PRIMARY⟩ ::= ⟨(_EXPRESSION⟩⟨LOGICAL_EXPRESSION⟩)
⟨RELAT⟩ ::= .EQ.
⟨RELAT⟩ ::= .LT.
⟨RELAT⟩ ::= .GT.
⟨RELAT⟩ ::= .NE.
⟨RELAT⟩ ::= .LE.
⟨RELAT⟩ ::= .GE.
⟨ARITH_EXPR⟩ ::= ⟨TERM⟩

```
⟨ARITH_EXPR⟩    ::= ⟨ARITH_EXPR⟩⟨+⟩⟨TERM⟩
⟨ARITH_EXPR⟩    ::= ⟨ARITH_EXPR⟩⟨-⟩⟨TERM⟩
⟨ARITH_EXPR⟩    ::= ⟨+⟩⟨TERM⟩
⟨ARITH_EXPR⟩    ::= ⟨-⟩⟨TERM⟩
⟨+⟩             ::= +
⟨-⟩             ::= -
⟨TERM⟩          ::= ⟨PRIMARY⟩
⟨TERM⟩          ::= ⟨TERM*⟩⟨PRIMARY⟩
⟨TERM*⟩         ::= ⟨TERM⟩*
⟨TERM⟩          ::= ⟨TERM/⟩⟨PRIMARY⟩
⟨TERM/⟩         ::= ⟨TERM⟩/
⟨PRIMARY⟩       ::= ⟨SOURCE⟩
⟨PRIMARY⟩       ::= ⟨PRIMARY_HEAD⟩⟨SOURCE⟩
⟨PRIMARY_HEAD⟩  ::= ⟨PRIMARY⟩**
⟨SOURCE⟩        ::= ⟨NO⟩
⟨SOURCE⟩        ::= ⟨OK_ID⟩
⟨SOURCE⟩        ::= ⟨(_EXPRESSION⟩⟨ARITH_EXPR⟩)
⟨SOURCE⟩        ::= ⟨FUNCT⟩⟨ARITH_EXPR⟩)
⟨FUNCT⟩         ::= SIN (
⟨FUNCT⟩         ::= COS (
⟨FUNCT⟩         ::= LOG (
⟨FUNCT⟩         ::= TAN (
⟨FUNCT⟩         ::= ATAN (
⟨FUNCT⟩         ::= INT (
⟨FUNCT⟩         ::= EXP (
⟨FUNCT⟩         ::= SQRT (
⟨FUNCT⟩         ::= ABS (
⟨OK_ID⟩         ::= ⟨ID⟩
```

Appendix B
Compiler Messages

The following are the error messages given by the SOL compiler. The messages are listed in alphabetical order. The type of message is described, along with the probable cause, and probable source of aid in this reference.

ASSEMBLAGE ILLEGALLY DEFINED IN IF OR DO STATEMENT
*** ERROR ***

— An **ASSEMBLAGE** CANNOT appear inside IF or DO statements.

. . . . . 6.3 (IF)
. . . . . 6.4 (DO)
. . . . . 7.1

ASSEMBLAGE OR COMPONENT SCOPE ERROR
*** FATAL ERROR ***

— This is an internal compiler error which indicates the compiler software is not functioning properly. This error should never appear.

CAREFUL, THIS MACRO IS BEING REDEFINED
*** WARNING ***

— You are using ?def or ?xdef with a macro that is already defined. This warning message appears in case the redefinition was accidental.

. . . . . C.1 (simple macros).
. . . . . C.2 (parametric macros).

COMPILATION ABORTED.
*** FATAL ERROR ***

— The compiler cannot recover from your errors, so the current compilation halts.

COMPONENT OR ASSEMBLAGE NOT ENDED CORRECTLY.
*** ERROR ***

— The same name must be used to start and end an **ASSEMBLAGE** or **COMPONENT** statement.

. . . . . 7.1

COMPONENT OUTSIDE ASSEMBLAGE OR INSIDE DO OR IF STMT.
*** ERROR ***

— **COMPONENT** statements are illegal outside an **ASSEMBLAGE** statement or within an **IF/THEN/ELSE** or **DO** statement.

. . . . . 6.3 (IF)

. . . . . 6.4 (DO)

. . . . . 7.1 (III) (ASSEMBLAGE)

CONSTRAINED OPTIMIZER WITH UNCONSTRAINED PROBLEM
*** ERROR ***

— Algorithms for constrained optimization, such as **Feasible Directions** or **Modified Feasible Directions**, CANNOT be used for unconstrained problems unless a strategy is used. You must either employ a strategy or switch optimization methods.

. . . . . 8.3.1 (Table 8–2)

DEPENDENT PARAMETER NOT INITIALIZED IN SUBROUTINE
*** ERROR ***

— Subroutine dependent parameters MUST be initialized within the subroutine.

. . . . . 9.1.2

DESIGN VARIABLES UNBOUNDED DURING NORMALIZATION
*** ERROR ***

— Unbounded design variables cannot be normalized. Either eschew normalization or add bounds to all design variables.

. . . . . 8.3.3

DIGIT EXCEEDS BASE
*** ERROR ***

— This number is too large and beyond the compiler's range. Typically occurs when scientific notation is used with an overly large exponent.

. . . . . 2.2.5

DUPLICATE ASSEMBLAGE DECLARED.
*** ERROR ***

— No more than a single **ASSEMBLAGE** can appear per main program or subroutine.

. . . . . 7.1 (II)

. . . . . 11.4

DUPLICATE IDENTIFIER DECLARED.
*** ERROR ***

— In general this error occurs whenever the same name is used to refer to two or more objects, for example a variable and a subroutine. Some common possibilities are listed below.

— You have given a subroutine the same name as one of its arguments

. . . . . 5.2

— You have given two subroutine formal parameters the same names.

. . . . . 5.2

. . . . . 9.1.2

— You have given two subroutines the same names

. . . . . 5.2

— You have declared a variable to be of two types in the declaration section.

. . . . . 5.1

— You have given two **COMPONENT**s at the same nesting level the same names.

. . . . . 7.1 (III).

— You have given a variable the same name as a subroutine or the main program.

. . . . . 5.2 (SUB)

. . . . . 11.1 (MAIN)

DUPLICATE OPTIMIZATION VARIABLE DECLARED.
*** ERROR ***

— Design variable and constraint names must be unique within a given **OPTIMIZE** statement.

. . . . . 8.1 (Design variables)

. . . . . 8.2 (Constraints)

EMITTED LINE EXCEEDS MAX CONTINUATIONS.
*** ERROR ***

— The **FORTRAN** emitted by the SOL compiler allows only 19 consecutive continuation lines using the & symbol. Break the offending SOL statement into several shorter SOL statements to avoid this problem.

EMPTY FILE PASSED TO INCLUDE MACRO
*** WARNING ***

— An empty file is being included, the rest of the line on which the `?include` macro appears is ignored

. . . . . C.4.3

EXPECTING APPEND TEXT, A { SYMBOL SHOULD BE HERE
*** ERROR ***

— When using the **?append** or **?xappend** macros, the next non-blank character after the macro name should be a {, to indicate the start of the replacement text.

. . . . . C.4.7

FILE OVERFLOW - MACRO NESTING TOO DEEP
*** ERROR ***

— In doing macro expansion, files are used. If more than 10 macros are being expanded at once (e.g. via **?xdef**), this error occurs. The error results from macros calling other macros calling other macros ... Reduce the nesting level (complexity) of your macro calls.

FORMAT OF THIS TYPE HAS NO DECIMAL POINT
*** ERROR ***

— You cannot use a decimal point with an I or L format

. . . . . 6.2.2.3 and 6.2.2.4

FORMAT OF THIS TYPE REQUIRES A DECIMAL POINT
*** ERROR ***

— You must use a decimal point with an E or F format

. . . . . 6.2.2.1 and 6.2.2.2

FORMAT TYPE ILLEGAL FOR INTEGER VARIABLES
*** ERROR ***

— You must use an I or L format when printing integer variables

. . . . . 6.2.2

FORMAT TYPE ILLEGAL FOR LOGICAL VARIABLES
*** ERROR ***

— You must use an L format when printing logical variables

. . . . . 6.2.2

FORMAT TYPE ILLEGAL FOR REAL VARIABLES
*** ERROR ***

— You cannot use an I or L format when printing real variables

. . . . . 6.2.2

FORTRAN BLOCK BEFORE MAIN PROGRAM BEGINS
*** ERROR ***

— You have started a FORTRAN block before your SOL program has begun. You should move the FORTRAN block after your program's header.

. . . . . 6.8

FORTRAN BLOCK DELIMITERS MUST BE IN COLUMN ONE
*** WARNING ***

— You should put your block delimiters in column one to be sure your FORTRAN code will be spaced correctly in the compiler's output.

. . . . . 6.8

FORTRAN BLOCKS ILLEGAL IN A COMPONENT OR ASSEMBLAGE
*** ERROR ***

— FORTRAN blocks cannot appear inside of ASSEMBLAGEs or COMPONENTs, because errors could result in the FORTRAN code emitted. The variables emitted in the FORTRAN code often are emitted with aliases. The FORTRAN block will not be accessing the same variables, and errors can result.

. . . . . 6.8

FURTHER INPUT IGNORED.
*** ERROR ***

— Self-explanatory; the compiler stops parsing the present line. This message often appears with other error messages.

GRAMMAR DOES NOT CONTAIN AN UNCONDITIONAL REDUCTION.
*** FATAL ERROR ***

— This is an internal compiler error that indicates the compiler software is not functioning properly. It should never appear.

ID MUST BE REAL, INTEGER OR LOGICAL FOR FORMAT
*** ERROR ***

— You have probably tried to print a subroutine name, ASSEMBLAGE or COMPONENT name or some other non-variable. Only variables can be printed.

IDENTIFIERS MUST BE LESS THAN 28 CHARACTERS LONG
*** WARNING ***

— You have used an identifier that is longer than 27 characters.

. . . . . 2.2.4

ILLEGAL ARGUMENT; REAL TYPE ARGUMENT IS REQUIRED
*** ERROR ***

— INTEGER or LOGICAL variables and/or expressions CANNOT be passed to functions which require a REAL argument.

. . . . . Chapter 10.

ILLEGAL CHARACTER
*** ERROR ***

— An illegal character which the SOL compiler does not recognize appears in your program.

. . . . . 2.2

ILLEGAL INITIALIZATION OF AN INDEPENDENT PARAMETER
*** ERROR ***

— Independent subroutine parameters CANNOT be assigned a value; only dependent parameters can be assigned values.

. . . . . 9.1.2, 9.2.1.1, and 9.2.3

ILLEGAL ITERATION: VARIABLE ALREADY INITIALIZED
*** ERROR ***

— **ASSEMBLAGE** or **COMPONENT** iteration variables CANNOT be initialized before the **ITERATION** section of the **ASSEMBLAGE** or **COMPONENT**.

. . . . . 7.3

ILLEGAL LIST CALL, ON AND OFF ARE THE ONLY SETTINGS
*** ERROR ***

— The **?list** macro with an has only two legal settings: **ON** and **OFF**.

. . . . . C.4.4

ILLEGAL MACRO USE: UNDEFINED MACRO DISCOVERED
*** ERROR ***

— Macros MUST be defined before they are used.

. . . . . Appendix C

ILLEGAL NAME RESULTS FROM CALL TO COMPONENT MACRO
*** ERROR ***

— Expansion of the **?component_name** macro created an overlong line. Try to break the original line up with the continuation symbol to avoid overlong expansion.

. . . . . C.4.6

ILLEGAL OPTIMIZER AND STRATEGY COMBINATION
*** ERROR ***

— Not all combinations of optimizer and strategy are legal.

. . . . . 8.3.1 (Table 8–4)

ILLEGAL SYNTAX IN FORMAT STATEMENT
*** ERROR ***

— An illegal character appears within the format statement

— The field width of your format statement is zero or less.

. . . . . 6.2.2

ILLEGAL USE OF AN EXTENDED IDENTIFIER
*** ERROR ***

— Extended identifiers CANNOT be used as subroutine names, **ASSEMBLAGE** or **COMPONENT** names, objective variable names for optimization, or in other ways; extended identifiers can only be used to refer to variables initialized inside of **ASSEMBLAGE**s or **COMPONENT**s.

. . . . . 7.1.2

ILLEGAL USE OF AN OPTIMIZATION DESIGN VARIABLE
*** ERROR ***

— Design variable cannot be used to define other design variables in the USE section of an OPTIMIZE statement, nor can values be assigned to a design variable with assignment statement or subroutine call.

. . . . . 8.1

ILLEGAL USE OF PREVIOUSLY INITIALIZED LOOP VARIABLE
*** ERROR ***

— The control variable of a DO loop cannot be altered within the loop. For example, the control variable of a nested DO loop cannot be the same as an outer loop's control variable.

. . . . . 6.4.1

ILLEGAL USE OF SUMMARIZATION EXPRESSION VARIABLE
*** ERROR ***

— Expression summarization variables are ONLY implicitly initialized at the end of an ASSEMBLAGE or COMPONENT; it is ILLEGAL to assign values to an expression summarization variable.

. . . . . 7.1.1.1

ILLEGAL USE OF ZERO LENGTH STRING
*** ERROR ***

— A null string CANNOT be used in specifying a COMPONENT or ASSEMBLAGE's summarize print information. Use a blank string (' ') instead.

. . . . . 6.2.3.3

INCORRECT PARSER TABLES.
*** ERROR ***

— This is an internal compiler error indicating that the SOL compiler is not working properly. It should never appear.

INPUT LINE EXCEEDS 120 CHARS.
*** ERROR ***

— SOL lines can be at most 120 characters long. If the line contains macros, perhaps the trouble is due to macro expansion. Use the continuation symbol, &, to break the line into smaller pieces.

. . . . . 2.4

INVALID DATA TYPE FOR USE.
*** ERROR ***

— A subroutine name, ASSEMBLAGE or COMPONENT name, or some other type of identifier appears where a variable MUST appear.

INVALID SYNTAX FOR REAL NUMBER.
*** ERROR ***

— Check the syntax of a REAL number.

. . . . . 2.2.5 and 3.3

IT IS ILLEGAL TO APPEND TO AN UNDEFINED MACRO
*** ERROR ***

— A macro must already be defined before the **?append** and **?xappend** macros can be used to add additional text.

. . . . . C.4.7

ITERATION VARIABLE HAS NOT BEEN INITIALIZED
*** ERROR ***

— **ASSEMBLAGE** and **COMPONENT** iteration variables MUST be initialized within the iterating **ASSEMBLAGE** or **COMPONENT**.

. . . . . 7.3

LAST ERROR ABORTED COMPILER.
*** FATAL ERROR ***

— Self-explanatory; the compiler is so wounded by the input program, it cannot go on.

MACRO ARGUMENTS DO NOT MATCH DEFINITION
*** ERROR ***

— the called macro's arguments do not match its defined pattern. Check the macro definition and call to see if they match.

. . . . . C.2

. . . . . C.3

MACRO PARAMETERS MUST BE SEQUENTIAL
*** ERROR ***

— Macro parameters must be defined in a sequential order, i.e., #2 appears before #3.

. . . . . C.2

MISSING (gives symbol inserted) INSERTED BEFORE SYMBOL.
*** ERROR ***

— The compiler could not parse the SOL code input, but made the change above and will try and go on.

MISSPELLED (gives symbol) CORRECTED.
*** ERROR ***

— The compiler could not parse the SOL code input, but guessed that a word was just misspelled, corrected it and will try to go on.

NO COMPONENTS HAVE ENDED
*** ERROR ***

— A **SUMMARIZE** print statement CANNOT appear until at least one **COMPONENT** or an **ASSEMBLAGE** has ended.

. . . . . 6.2.3.1

NO DESIGN VARIABLES DEFINED IN OPTIMIZATION
*** ERROR ***

— At least ONE design variable MUST appear within every `OPTIMIZE` statement.

. . . . . 8.1

NO FILE NAME GIVEN.
*** FATAL ERROR ***

— No SOL code source file has been given, so there is nothing to compile

. . . . . 1.1

NO RULE CAN BE APPLIED IN PARSER.
*** ERROR ***

— Internal compiler error indicating that the compiler software is not functioning properly. This error should never appear.

NO SUMMARIZATION VARIABLES HAVE BEEN DECLARED
*** ERROR ***

— At least one summarization variable MUST be declared in an `ASSEMBLAGE` statement.

. . . . . 7.1.1

NUMBER REPRESENTATION GREATER THAN 27 CHARACTERS
*** ERROR ***

— Numbers CANNOT be longer than 27 characters.

. . . . . 2.2.5

OPTIMIZATION VARIABLE IS UNINITIALIZED
*** ERROR ***

— The objective function or a constraint variable for an `OPTIMIZE` statement has been left uninitialized. These variables must be initialized within the `OPTIMIZE` statement (i.e. by assignment statement or subroutine call.)

. . . . . Chapter 8 and 8.2

OUT OF INFO WHILE COLLECTING MACRO
*** ERROR ***

— The SOL program ended before the last macro call was completed.

. . . . . Appendix C

OUTER SCOPE VARIABLE ALTERED IN ASSEMBLAGE OR COMPONENT
*** WARNING ***

— `ASSEMBLAGE`s and `COMPONENT`s can alter non-local variables; this warning flags the alteration in case it was accidental.

. . . . . 7.2

OVERLONG LINE IN MACRO DEFINITION
*** ERROR ***

— Delimited macro's CANNOT have a pattern more than 120 characters long. The continuation symbol CANNOT be used here and a smaller pattern must be used.

. . . . . C.3

OVERLONG LINE IN MACRO EXPANSION
*** ERROR ***

— A line in your macro call expands beyond line size, 120, when the parameters are substituted. Shorten the line by altering the macro definition.

. . . . . C.2

PROGRAM NOT ENDED CORRECTLY.
*** ERROR ***

— The name at the start of a SOL program MUST be the same as the name used to end it.

. . . . . 2.1.5

REAL NUMBER EXCEEDS MACHINE RANGE.
*** ERROR ***

— SOL numbers are limited in range.

. . . . . 2.2.5, 3.1 and 3.3

SEVERE SYNTAX ERROR
*** ERROR ***

— An illegal character or word appears in your SOL program

SEVERE SYNTAX ERROR CAUSED PARSER FAILURE.
*** FATAL ERROR ***

— The compiler cannot recover from the previous syntactic errors, and will stop execution. Fix the errors found and recompile.

SOURCE PROGRAM FILE IS EMPTY.
*** FATAL ERROR ***

— An empty file; there is no program to compile so execution halts.

. . . . . 1.1

STRING EXCEEDS ALLOWABLE STRING LENGTH
*** ERROR ***

— Strings CANNOT be longer than 61 characters.

SUBROUTINE ARGUMENT DEPENDENCY NOT MATCH DECLARATION
*** ERROR ***

— The number of independent or dependent parameters in the subroutine implementation does not match the subroutine's declaration. The total number of parameters is correct, but there are too many or too few dependent/independent parameters

. . . . . 9.1.2

SUBROUTINE ARGUMENT NUMBER NOT MATCH DECLARATION
*** ERROR ***

— The number of parameters in the subroutine implementation does not match the subroutine's declaration.

. . . . . 9.1.2

SUBROUTINE ARGUMENT TYPES NOT MATCH DECLARATION
*** ERROR ***

— The argument types in your subroutine implementation do not match the types of arguments given in the subroutine's declaration.

. . . . . 9.1.2

SUBROUTINE DEFINED TWICE IN SOURCE
*** ERROR ***

— You have two implementations of the same subroutine; eliminate one.

SUBROUTINE NAME NOT DECLARED
*** ERROR ***

— The implemented subroutine was not declared.

. . . . . 9.1.1 and 9.1.2

SUBROUTINE UNDEFINED IN SOURCE
*** ERROR ***

— The declared subroutine is never implemented.

. . . . . 9.1.1 and 9.1.2

SUMMARIZATION VARIABLE DECLARED AS A NON-REAL TYPE
*** ERROR ***

— Summarization variables and summarization expression variables MUST be **REAL**; change the **DECLARE** section or rename your summarization variable.

. . . . . 5.1

. . . . . 7.1.1.1

SUMMARIZATION VARIABLES INITIALIZED BEFORE ASSEMBLAGE; LOCAL COPY MADE
*** WARNING ***

— The summarization variable declared has already been intialized; a local copy will be used within the **ASSEMBLAGE**.

. . . . . 7.1.1.1 and 7.2

SUMMARIZATION VARIABLES INITIALIZED WHEN SUB-COMPONENTS EXIST
*** ERROR ***

— The summarization variables of an **ASSEMBLAGE** or **COMPONENT** which contain nested **COMPONENT**s CANNOT be initialized; such variables are initialized automatically.

. . . . . 7.1.1.1

SUMMARIZATION VARIABLES UNINITIALIZED
*** ERROR ***

— The summarization variables of an **ASSEMBLAGE** or **COMPONENT** which DO NOT contain nested **COMPONENT**s MUST be initialized.

. . . . . 7.1.1.1

SYNTAX DEMANDS A ? SYMBOL HERE
*** ERROR ***

— The compiler expects a macro name to appear, but you have not supplied a leading macro symbol,?.

. . . . . C.1

THE INCLUDE MACRO MUST APPEAR ALONE ON A LINE
*** ERROR ***

— Other non-blank characters CANNOT appear on the same line as the `?include` macro.

. . . . . C.4.3

THE INPUT FILE NAME IS TOO LONG.
*** ERROR ***

— File names cannot be longer than 120 characters.

THIS IS AN ILLEGAL OPTIMIZATION SWITCH
*** ERROR ***

— This is not a valid optimizer option setting.

. . . . . 8.3 (8.3.3 and 8.3.4 esp.)

TOO MANY ITEMS IN PRINT LIST, ONLY 20 ALLOWED
*** ERROR ***

— At most 20 items can be printed with a single **PRINT** or **SUMMARIZE** print statement.

. . . . . 6.2.1 and 6.2.3

UNCONSTRAINED OPTIMIZER WITH A CONSTRAINED PROBLEM
*** ERROR ***

— No constraints can be used with this optimization algorithm unless a strategy option is chosen. You must either employ a strategy or switch optimization methods.

. . . . . 8.3.1 (Table 8–4)

UNEXPECTED SYMBOL DELETED.
*** ERROR ***

— The compiler cannot parse your program, and has deleted the above symbol and will try and go on.

UNEXPECTED SYMBOL REPLACED BY (gives replacement symbol)
*** ERROR ***

— The compiler cannot parse your program, and has replaced an unexpected character or word with the above, and will try and go on

UNIMPLEMENTED SUBROUTINE(S)
*** ERROR ***

— The subroutines listed were declared BUT NOT implemented. Either alter the **DECLARE** section or implement the subroutines.

. . . . . 5.2

. . . . . 9.1.2

UNINITIALIZED IDENTIFIER.
*** ERROR ***

— Variables cannot be accessed until they are initialized

VALUE EXCEEDS MAXIMUM INTEGER.
*** ERROR ***

— **INTEGER**s have a limited range.

. . . . . 3.1

VARIABLE NOT DECLARED AS A SUMMARIZATION VARIABLE
*** ERROR ***

— **SUMMARIZE** print statements are legal only with summarization variables.

. . . . . 6.2.3

Appendix C
SOL Macros – Advanced Material

Macros are an advanced feature of SOL and are not recommended for the novice. An experienced SOL user can benefit from the use of macros, especially when developing large SOL programs.

Macros are like abbreviations. A single character, or short word, can represent longer sequences of text that are used repeatedly. The simplest macro consists of a macro name and a body of replacement text. Defining a macro involves associating a name with replacement text. When a macro name appears in a SOL program, the replacement text is substituted for the name at COMPILE-TIME. This act of replacement is known as a macro call or macro expansion. Macros allow text substitution in a SOL program.

This may seem complex, but a simple example will clarify the concept. Suppose we define a macro named `?pi` (all macros begin with the ? symbol), with the replacement text "3.141592654," at the start of a SOL program. Then, whenever the name `?pi` appeared in my SOL program, it would be replaced with 3.141592654. For instance, `x = 2 * ?pi * r` would become, `x = 2 * 3.141592654 * r`

Naturally, macro definitions are not generally used just to speed up the typing of one isolated formula. The real advantage lies in using a macro abbreviation for clusters of code or text that are used dozens of times throughout a SOL program.

Abbreviations like `?pi` are useful in many applications, and they are powerful. One little macro can represent an enormous amount of material. The judicious use of macros can reduce or eliminate the tedium of retyping repetitive portions of identical or similar code. Further, a macro can be made an abbreviation for a complex tangle of code. By giving the macro a carefully selected name which reflects the function of the code, the macro can be made more understandable than the text that will be substituted. In this way, a tidy, descriptive macro call can appear in the SOL program, and the messy code is hidden away as replacement text.

- It is important to note that the text replacement occurs at COMPILE-TIME, and is not a dynamic run-time event.

SOL offers four different kinds of macros:

C.1 SIMPLE MACROS

Simple macros consist in a macro name and a body of replacement text. Using macros involves two elements:

1) Defining the macro and its replacement text
2) Calling (expanding) the macro

In order to use a macro, it MUST be defined first. SOL provides the means to define macros by using a predefined macro, **?def** (All macros begin with a ? symbol). A simple macro definition has the following syntax:

?def ⟨ macro_name ⟩ { ⟨ replacement text ⟩ }

where:

⟨ macro name ⟩ is the name of the macro. It must consist of a ? symbol, followed by either a legal SOL identifier or a single non-alphabetic character.
- Extended Identifiers are not legal macro names.

⟨ replacement text ⟩ is the body of replacement text. Any characters can appear in the replacement text.
- However, additional restrictions apply to the use of open brace, {, and close brace, }, within the replacement text. See section C.1.2 of this appendix for details.

There are a number of stipulations for proper use of SOL macros. These stipulations fall naturally into two categories:

C.1.1 GENERAL RULES FOR SIMPLE MACRO DEFINITIONS

The following restrictions apply to simple macro definitions:

1) No spaces can appear between the ? symbol and the word, **def** or between the ? symbol and the first character of the macro name.
2) Zero or more spaces, tabs and/or carriage returns are allowed between the ⟨ macro name ⟩ and the open bracket, {, that signifies the start of the macro's replacement text.
3) Macro definitions can appear anywhere a blank line can appear in a SOL program, including before the main program header (e.g., **PROGRAM test**) appears.
4) More than one macro definition can appear per line, although one per line is recommended for readability's sake.

5) A macro must be defined before it is called.

6) Macros CAN be redefined by subsequent macro definitions. The most recent macro definition holds.

- SOL gives a warning message announcing that a macro is being redefined, just in case the redefinition is accidental.

To illustrate these points, consider the following macro definition in the context of a SOL program (annotated with line numbers to facilitate the discussion):

```
A01 PROGRAM test
A02 ?def ?incr_x        {x = x + 3.141592654}
A03
A04 x = 0
A05 DO i = 1, 10
A06   ?incr_x
A07   print x
A08   ?incr_x
A09   print x
A10 ENDDO
A11
A12 END test
```

The macro definition appears on line A02. It associates the macro name, `?incr_x`, with a body of replacement text, `x = x + 3.141592654`. Wherever the name `?incr_x` is found, it will be replaced by the SOL compiler with the replacement text, `x = x + 3.141592654`. Thus, lines A05 – A10, are seen by the SOL compiler EXACTLY AS IF the following was typed:

```
B05 DO i = 1, 10
B06  X = X + 3.141592654
B07   print x
B08  X = X + 3.141592654
B09   print x
B10 END DO
```

Notice that the macro names have all been replaced by the replacement text. Calls to user defined macros (e.g., A06 or A08) do not appear in the SOL compiler LISTING file (See Chapter 1, section 1.4). Rather, the expanded text appears instead.

As a further example, consider the following sample program:

```
A01 ?def ?prt_incr_x
A02 {print x
A03 x = x + 2}
A04 ! some macro definitions before the main program
A05 ! begins.  Notice that the replacement text stretches
A06 ! over two lines
A07 ?def ?pythag_x
A08 { x = sqrt((side1 ** 2) + (side2 ** 2))
A09 ?prt_incr_x  }
A10 Program x_printer
A11   x = 12
A12   ?prt_incr_x
A13   side1 = 3
A14   side2 = 4
A15   ?pythag_x
A16   print x
A17 end x_printer
```

In this example, the macros on lines A12 through A16 are expanded, and the SOL compiler acts EXACTLY as if the following was typed instead of lines A12 through A16:

```
B12   PRINT X
B13   X = X + 2
B14   side1 = 3
B15   side2 = 4
B16   X= SQRT( (SIDE1 ** 2) + (SIDE2 ** 2))
B17   PRINT X
B18   X = X + 2
B19   print x
```

The macro on line A12 is expanded into lines B12 through B13. The macro on line A15 is expanded on lines B16 through B18. Notice that the call to `?ptr_incr_x` is also expanded.

These examples are fairly simple, but illustrate the basic principle behind simple macros.

C.1.2 Simple Macro Definition ⟨ replacement text ⟩

The following restrictions hold for the syntax and body of the replacement text:

1) A macro's replacement text CAN include calls to other macros.

2) A macro CANNOT call itself in the replacement text of its own definition, either directly or indirectly. SOL DOES NOT PROVIDE ERROR-CHECKING FOR THIS, so you must avoid this condition on your own.

3) Braces, { and }, CANNOT be included in the replacement text UNLESS they are paired.

- This restriction requires a more detailed explanation. The replacement text is delimited by an open brace, {, and a close brace, }. But consider the following declaration: (columns denoted shown for convenience)

```
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN
    ?def ?macro_1 {replace} me }
```

The SOL compiler cannot determine whether the brace in column t is intended to end the replacement text, or whether the end of the text occurs in column y. However, it is useful to have braces appear in macro replacement texts, so that macro definitions can appear in the replacement text, for example:

```
?def ?Macro_1 { ?def macro_2 {a = 6} }
```

- Any open brace in replacement text is matched with the nearest unpaired close brace, so only pairs of braces can appear in the replacement text.
- The replacement text will be substituted for any macro call. The compiler will act just as if the replacement text appeared in the program, instead of the macro call.
- The listing file will contain the "expanded" replacement text, instead of the original call.

C.2 PARAMETRIC MACROS

A useful addition to the simple macro would be the capability to have macros in which some of the replacement text is changeable; the replacement text would become a template, filled in with different things when the macro is used. SOL offers this capability with its parametric macros. Macros can be defined in terms of parameters, and arguments are supplied when the macro is used; the arguments are substituted for the parameters in the replacement text.

This section is divided into two sections:

C.2.1 - Explains how to define parametric macros

C.2.2 - Explains how to use parametric macros

C.2.1 PARAMETRIC MACRO DEFINITION

Before a macro can be used, it must be defined, and parametric macros are no exception. Defining parametric macros is very much like defining simple macros, except "blanks" are specified in the replacement text that will be filled in when the macro is used, and the way these "blanks" receive values is also specified. A parametric macro has the following syntax:

?def ⟨ macro_name ⟩ ⟨ parameters ⟩ { ⟨ replacement text ⟩ }

where:

⟨ macro name ⟩ — is the name of the macro. It must consist of a ? symbol, followed by a legal SOL identifier or a single non-alphanumeric character.

- Extended Identifiers are not legal macro names.

⟨ parameters ⟩ — is the macro's parameter list specifying the "blanks" that will be filled in later. The parameter list consists of a series of # ⟨ digit ⟩ pairs, (e.g., #1 or #4), where the ⟨ digit ⟩ is a number between 1 and 9.

⟨ replacement text ⟩ — is the body of replacement text. Any characters can appear in the replacement text. Parameters, such as #1 appear where variable replacement text is desired.

- However, additional restrictions apply to the use of open brace, {, and close brace, }, within the replacement text. See section C.1.2 of this appendix for details.

A parametric macro definition takes the form of a template, leaving holes to be filled in later. This can best be explained with an example.

Suppose a macro is needed that would produce code to cube the variable, **x**. This is a simple macro, and might be defined as below:

```
?def ?cube {x = x * x * x}
```

However, after a while it becomes clear that it would be useful to have a more general macro that would produce code to cube any variable. Such a parametric macro might be defined as below:

```
?def ?cube #1 {#1 = #1 * #1 * #1}
```

With this definition, `?cube x` would expand to `x = x * x * x` and `?cube y` would expand to, `y = y * y * y`. The symbol #1 stands for the first parameter to the macro, and when you say `?cube x`, `x` is the so-called argument that will be substituted for #1 in the replacement text.

As the notation, #1, suggests, macros can have more than one parameter. There can be as many as nine parameters, #1 to #9, and they must be numbered in order. For example, #4 cannot be used in a definition, unless the previous parameter was #3.

The restrictions and regulations governing the definition of parametric macros are given below:

1) There can be no more than nine parameters, starting with #1 and increasing to #9.

2) Parameters in a definition must be specified in order. So for example, a macro with four parameters, MUST name its parameters #1 to #4, in that order.

3) Restriction 2) applies only to the definition of parameters (before the replacement text begins). Parameters can appear in any order desired within the replacement text.

4) No spaces can appear between the "#" symbol and the number, "1" to "9."

5) Blanks, tabs and carriage returns are IGNORED between the macro name and the first parameter, between the parameters and between the last parameter and the start of the replacement text, i.e. the following are equivalent:

```
i.    ?def ?mac1#1#2{...}
ii.   ?def ?mac1
      #1#2 {...}
iii.  ?def ?mac1        #1#2 {...}
iv.   ?def ?mac1 #1     #2     {...}
v.    ?def ?mac1 #1 #2 {...}
vi.   ?def ?mac1 #1
      #2 {...}
vii.  ?def ?mac1 #1#2{...}
```

6) Further, parametric macros abide by the same regulations as simple macros, as stipulated in section C.2 of this chapter.

C.2.2 Parametric Macro Use

Once a parametric macro has been defined, it can be used in a SOL program. Parametric macros are used the same way as simple macros, except that the actual "arguments" to the parameters must be supplied when the parametric macro is used. These actual arguments are associated with the macro's parameters. Everywhere the parameter appears in the replacement text, the associated actual argument is substituted. The replacement text, with substitutions, replaces the macro call in the SOL program.

Parametric macro calls have the following syntax:

? ⟨ name ⟩ ⟨ argument list ⟩

where:

⟨ name ⟩	is the name of the macro. Nothing can appear between the symbol, ?, and the name of the macro. The name is any legal SOL identifier.
⟨ argument list ⟩	is the list of arguments. The exact syntax of the argument list is discussed in the next section, C.2.2.1

For example, the following are syntactically legal calls to parametric macros:

— `?cube x`

— `?mac1 a b`

— `?mac1 a b`

— `?rectangle_area 14 12 total_area`

This section is divided into two sections:

C.2.2.1 – Discusses what constitutes an argument

C.2.2.2 – Discusses how arguments and parameters are associated.

C.2.2.1 Arguments to Parametric Macros

An argument to a parametric macro can be any of the following:

1) a SOL identifier

   E.g., the_name, abcde a2345_7@b

2) a SOL string

   E.g., 'the string' , 'Walter Cronkite: what a guy' , 'Cheap Hotel'

- **Note:** Macro calls will NOT be expanded inside strings. However, quote marks are stripped off and WILL NOT appear in the replacement text, (e.g. see C.2.2.2, (example 1, 3)) for an example).

3) a signed (+, -) or unsigned number

   E.g., .0992 , -4 , +3.456 , 8

4) # ⟨ digit ⟩ , where ⟨ digit ⟩ is a number, "1".."9", no spaces between # and ⟨ digit ⟩ .

   E.g., #2 , #8 , #1 (See **example 1** item 2) in section C.2.2.2).

5) any character other than a blank or tab character, a ?, or the quote symbol, '.

   E.g., #, %, ^ , y, g, U, i, * -,), 0, 4, 6, {.

These arguments are combined to form an ⟨ argument list ⟩ for a parametric macro call. An ⟨ argument list ⟩ to a parametric macro must abide by the following regulations:

1) If the argument is an identifer, unsigned number, or a character that can appear in a SOL identifier, one or more tabs, blanks and/or carriage returns must separate the argument from the macro name, otherwise, zero or more blanks, tabs and carriage returns can appear between the argument and the macro name. For example:

   DEFINITIONS:

```
?def ?mac1 #1 #2 {  }
?def ?mac2_a #1 #2 {  }
?def ?mac5 #1     #2 {  }
```

```
?def ?mac4 #1 #2 #3 {  }
```

Legal	*Illegal*
`?mac1 12 *`	`?mac112 *`
`?mac2_a alfa %`	`?mac2_aalfa %`
`?mac5 _ 65`	`?mac5_ 65`
`?mac3+4.3 Walter`	
`?mac4`	
`a`	
`b`	
`c`	

- The SOL compiler cannot distinguish between the name and the argument in the case of arguments that are identifers, numbers, or characters that can appear in SOL identifiers, UNLESS at least one blank or carriage return separates them.

2) Arguments are separated with zero or more blanks/tabs and carriage returns, with the following exceptions:

   — two identifier arguments must be separated by at least one blank, tab or carriage return.

   — two numerical arguments, when the second argument is unsigned, must be separated by at least one blank, tab or carriage return.

   — An identifier argument must be separated by one or more blanks or carriage returns from unsigned numerical arguments, or single character arguments, when the character can appear in a legal SOL identifier.

For example, given the definition, `?def ?mac1 #1 #2 #3`, the following macro calls are equivalent:

```
1)  ?mac1 ident 12.3 $

2)  ?mac1              ident     12.3$

3)  ?mac1                                ident
                 12.3
$

4)  ?mac1
            ident
            12.3
             $
```

The next section details how these arguments are associated with the parameters in parametric macros.

The association between arguments and macro parameters is simple: the first argument is associated with parameter #1, the second with parameter #2 and so on. Once all the parameters are matched, the macro is expanded. Just be careful to supply the same number of arguments as there are parameters, otherwise one of the following will occur:

1) **Too few arguments**: the SOL compiler will gobble up part of your SOL program for use as a parameter argument.

2) **Too many arguments**: the SOL compiler will interpret the extra arguments as part of the SOL program, and syntactic errors will probably result.

EXAMPLES:

Example 1:

Definition:

```
?def ?example #1 {#1}
```

Use:

1) `?example x`

This call will expand to be: `x`

2) `?example #1`

This call will expand to be: `#1`

3) `?example 'this is a string'`

This call will expand to be: `this is a string`

- Note that the quote marks are stripped off, you CANNOT put quote marks inside a string. IF quote marks were desired, the following definition could be used:

```
?def ?example #1 {'#1'}
```

- Strings must appear on a single line.

4) `?example`
`12.3`

This call will expand to be: `12.3`

5) `?example @`

This call will expand to be: `@`

Example 2:
Definition:

```
?def ?example2  #1
             #2    {#1 = #2.33
                    print #1}
```

Use:

```
1)     ?example2 a 13
```

This call will expand to be:

```
a = 13.33
print a
```

In the next section, parametric macros with delimited parameters, and some more complex uses of macros are discussed.

C.3 DELIMITED MACROS

SOL also allows you to provide "delimiters" for both simple and parametric macros. Delimiters consist of additional text, a pattern, that must be matched when the macro is called. This section is divided into two sections that provide detailed explanations:

C.3.1 – Discusses delimited simple macros

C.3.2 – Discusses delimited parametric macros

C.3.1 DELIMITED SIMPLE MACROS

Delimiters are specified when you give a macro definition. Delimited simple macros have the following syntax:

?def ⟨ name ⟩ ⟨ pattern ⟩ { ⟨ replacement text ⟩ }

where:

⟨ name ⟩	is the name of the macro being defined.
⟨ pattern ⟩	is the string of delimiters. This consists of one or more characters. All characters are legal except the following: ?, {, the blank space, and a tab. Any other character can be used as a delimiter.
⟨ replacement text ⟩	is the macros replacement text.

Delimited simple macros are best explained by example. Consider the following macro definition:

```
?def ?mac_1 pattern {x = 2}
```

This macro could be called in the following ways:

Call	*Expands to be*
`?mac_1 pa tt e rn`	`X = 2`
`?mac_1 pattern`	`X = 2`

When the macro is called the pattern, "p,a,t,t,e,r,n" must be matched before the macro will be replaced. If the pattern cannot be matched, an error results. For example, given the definition above, the following macro calls would be ILLEGAL:

— `?mac_1`

— `?mac_1 p at`

These are illegal because the entire pattern, "p,a,t,t,e,r,n" does not appear. When matching delimiters, blanks, tabs, and carriage returns are ignored; matching is done on a character by character basis. Thus, the definition above specifies that the first character is a "p," followed by an "a," followed by a "t," and so on. For example, the following are legal calls to ?mac_1:

— `?mac_1 pattern`

— `?mac_1 p a t t er n`

The precise rules that apply to the use of delimiters with simple macros are given below:

1) A delimiter can be any character except for the following: the macro symbol "?", the blank space " ", open brace "{", and a tab. A carriage return CANNOT be a delimiter.

2) The symbol,"#", has special significance. If the character immediately following the # is a digit in "1" ... "9", the #, digit pair is considered to denote a parameter, otherwise the character "#" is considered to be a delimiter.

3) The macro call must match the order and number of delimiters specified in the macro definition. If the call does not match, a SOL error will occur.

4) Blank spaces, tabs and carriage returns between delimiters are ignored, except in the case of "#" as detailed previously.

5) Delimiters are case sensitive. For example, `a` will not match `A`.

6) The pattern CANNOT be longer than 120 characters, the maximum length of a SOL input line.

   - The continuation symbol, &, cannot be used to dodge the length restriction.

In general, delimited simple macros are not very useful. Delimiters are primarily used for delimited parametric macros. However, delimited simple macros are needed to have prefix macros, as in the next example:

Example:

Suppose we want a macro that will expand as a prefix in forming arbitrary file names. Consider the following:

Definition:

```
?def ?prefix {name_of_file}
```

Use:

```
?prefix _obj
?prefix_list
```

Expansion:

```
name_of_file _obj
***ERROR, undefined macro, ''prefix_list''
```

As you can see, a simple macro will not do for a prefix macro; either a blank space appears between the expanded text and the root word (_obj), or the root is misinterpreted as part of the macro name, as in "prefix_list" Delimited simple macros solve this problem, as shown with the following definition:

Definition:

```
?def ?prefix :  {name_of_file}
```

Use:

```
?prefix :_obj
?prefix:_list
```

Expansion:

```
name_of_file_obj
name_of_file_list
```

Here, the delimiter, :, forces the `?prefix call` to match the delimiter before substituting the replacement text. In this way, macros which can be used as prefixes are possible.

Section C.5 offers a summary of the rules that govern macros in general.

C.3.2 Delimited Parametric Macros

Delimited parametric macros are treated in much the same way as delimited simple macros. However, delimited parametric macros can have parameters intermingled with the delimiters. Delimiters are specified when a macro is defined. The delimiters and the parameters are matched when the macro is called.

Delimited parametric macros have the following syntax:

?def ⟨ name ⟩ ⟨ pattern ⟩ { ⟨ replacement text ⟩ }

where:

⟨ name ⟩	is the name of the macro being defined.
⟨ pattern ⟩	is the string of delimiters and parameters. The delimiters consist of one or more characters. All characters are legal except the following: ?, {, the blank space, and tabs. Parameters are specified as usual with a # digit pair. (see C.2 for a full explanation of parameters).
⟨ replacement text ⟩	is the macros replacement text.

Delimited parametric macros are best explained with an example. Consider the following:

```
?def ?increment #1 by #2 from #3 to #4
{DO i = #3,#4
  #1 = #1 + #2
ENDDO }
```

This macro could be called in the following ways:

1) `?increment x by 3 from 5 to 9`

2) `?increment list by 1 from 1 to maximum_list`

The macro's arguments are determined as follows: the first argument is associated with #1, then a pair of characters `b` and `y` should be seen, then the next argument is associated with #2 , then the characters `f`, `r`, `o` and `m` should be seen, and so on. Thus, the two calls above would expand into the following:

1)
```
DO i = 5,9
  X = X + 3
ENDDO
```

2)
```
DO i = 1,MAXIMUM_LIST
  LIST = LIST + 1
ENDDO
```

Of particular note is the fact that blanks and carriage returns are ignored when matching up with delimiters. Thus, both of the following are legal calls to `?increment`:

— `?increment z b y 2 f r o m 1 to fast_time`

— `?increment z by 2 from 1 to fast_time`

The precise rules that govern the use of delimiters with parametric macros follow:

1) A delimiter can be any character except for the following: the macro symbol ?, the blank space " ", open brace "{", and tab. A carriage return cannot be a delimiter.

2) The symbol,"#", has special significance. If the character immediately following the # is a digit, the #, digit pair is considered to denote a parameter, otherwise the character "#" is considered to be a delimiter.

3) The macro call must match the order and number of parameters and delimiters specified in the macro definition. If the call does not match, a SOL error will occur.

4) Blank spaces, tabs, and carriage returns between delimiters are ignored, except in the case of "#" as detailed previously.

5) Delimiters are case sensitive. For example, a will not match A.

EXAMPLES:

Example 1:

Definition:

```
?def ?example ab#1 1:# 2 {print #1}
```

Use:

```
?example ab 12 1:# 2
?example ab12 1:#2
?example a b 12
1 :  # 2
```

All of the above are legal calls of **?example**. Note that the blank space in the definition between "#" and "2", make "#" and "2" delimiters, rather than denote parameter #2. In all three cases, #1 is associated with 12.

Bad Use:

```
?example ab 121:# 2
```

This is an illegal call to **?example**. Parameter #1 will be associated with the number, 121, and the delimiter, "1", will not be found.

Example 2:

Definition:

```
?def ?pattern { x :  } ?def ?example ?pattern #1, y :  #2 {print #1}
```

Because of the expansion of the macro, **?pattern**, the above definition of **?example** is equivalent to:

```
?def ?example x :  #1, y :  #2
```

Use:

```
?example x:134,y:ppop
?example x :  axis ,
y :  123.33
```

Section C.5 of this chapter offers a detailed summary of rules that govern macros in general.

C.4 PREDEFINED MACROS

SOL offers eight predefined macros:

C.4.1 THE ?DEF MACRO

The `?def` macro is an essential part of SOL macros, because it allows the SOL programmer to define his/her own macros. The `?def` macro has already been discussed tangentially in sections C.2 and C.3 of this chapter. The `?def` macro has the following syntax:

`?def` ⟨ macro name ⟩ ⟨ pattern ⟩ { ⟨ replacement text ⟩ }

where :

⟨ macro name ⟩	is the name of the macro and consists of the symbol, ?, followed immediately by a SOL identifier.
⟨ pattern ⟩	is the pattern text and consists of a series of delimiters and parameters. This part is optional and does not have to appear.
⟨ replacement text ⟩	is the replacement text, can be any characters, with stipulations on the use of braces. This section is delimited by the open and close braces, "{" and "}."

This has already been discussed in detail in sections C.1 – C.3 of this chapter. The following rules govern how the `?def` macro can be used in a SOL program:

1) A `?def` macro can appear anywhere in a SOL program blanks can appear. However, a `?def` cannot immediately appear after an `?include` macro, on the same line as the `?include` macro.

2) A call to the macro being defined CANNOT appear in the replacement text of the definition, e.g. `?def ?mac1 {?mac1}` is ILLEGAL, no error message is given, and an infinite loop generally results. In other words, this is BAD, don't do it. Also, don't do things like the following either:

```
?def ?a {6}
?def ?b {?a}
?def ?a {?b} !  As BAD as calling ?a directly.
```

3) Macro calls in the replacement text are not expanded until the macro being defined is CALLED. (See section C.4.2 of this appendix).

- If such an expansion is desired, use the **?xdef** macro.

4) Macros inside of comments are ignored, just as any other SOL statements.

C.4.2 The ?XDEF Macro

The **?xdef** macro acts exactly like the **?def** macro, but the replacement text is handled somewhat differently. In an **?xdef**, all macros in the replacement text are expanded when the macro is DEFINED. With a **?def**, all macros in the replacement text are expanded when the defined macro is CALLED. This difference can best be demonstrated with an example. Consider the following:

Example 1

```
?def ?mac1 {6}
?xdef ?mac2 {x = ?mac1}
?def ?mac1 {9}
?mac2
```

Example 2

```
?def ?mac1 {6}
?def ?mac2 {x = ?mac1}
?def ?mac1 {9}
?mac2
```

In example 1, **?mac1**, in **?mac2**'s replacement text, is expanded when **?mac2** is defined. Thus, the **?xdef** on the second line is equivalent to:

```
?def ?mac2 {x = 6}
```

When **?mac2** is called on the last line, it expands to be: **x = 6**.

However, in example 2, **?mac1** is expanded when **?mac2** is called, on the last line. By this time, **?mac1** has been redefined. Therefore, the call to **?mac2** in example 2 expands to be: **x = 9**.

Thus, a call to **?xdef** is equivalent to a call to **?def** except that the macros in the **?xdef**'s replacement text are expanded. Aside from the different expansion times of macros in the replacement text, **?xdef** and **?def** are equivalent.

C.4.3 The ?INCLUDE Macro

The **?include** macro is one of the most useful features of SOL, because it allows external files to be included in a SOL program. The **?include** macro has the following syntax:

?include ⟨ file name ⟩

where :

⟨ file name ⟩ is the name of the file to be included.

The **?include** macro call is replaced with the text of the included file in the LISTING. The following restrictions hold for the **?include** macro:

1) The ⟨ file name ⟩ portion of the **?include** macro must appear on the same line as the call to **?include**, or a SOL error results.

2) The file named ⟨ file name ⟩ must exist in the current directory, or the SOL compilation will ABORT while trying to open the file. The full path name can be given as a ⟨ file name ⟩ , but if that file does not exist, the SOL compilation will ABORT.

3) If the file is empty, a SOL compile-time warning will be issued, but compilation will continue.

4) The `?include` macro MUST appear on a line alone in a SOL program. However, an `?include` macro can be followed by a comment, but NO other statements, not even macros.

The `?include` macro is especially useful for introducing often used macros. By storing the macro definitions in a file, many SOL programs can use the same definitions by `?including` the definition file at the start of the programs. In the same way, useful subroutines can be stored in files, and included for use in other SOL programs. In this case, remember to `?include` the subroutine declarations, as well as implementations.

C.4.4 THE ?LIST MACRO

The `?list` macro allows the LISTING option to be turned on or off from within the SOL program being compiled. This is useful when only a few particularly significant portions of a SOL program are desired in the compilation listing. The `?list` macro has the following syntax:

`?list` ⟨ option ⟩

where:

⟨ option ⟩ can be one of two values: on or off. Thus to turn the listing on, type `?list on`, and to turn the listing off type, `?list off`. In a way, the `on` or `off` can be considered an argument to the `?list` macro.

The `?list` macro must abide by the following regulations:

1) No carriage returns can appear between the `?list` and its option, `on` or `off`. Blanks can appear.

2) A call to `?list` can appear anywhere in a SOL program blanks can appear.

   - A call cannot appear after an `?include` macro, on the same line as the `?include`.

3) All calls to `?list on` will appear in the listing file.

4) All calls to `?list off` will NOT appear in the listing file, because the macro turns the listing off.

5) The listing CANNOT be turned on, by a call to `?list on`, if the SOL program is compiled with the compiler LIST option OFF. Thus, in order to turn the listing on or off from within the compiled program, the SOL program must be compiled with the compiler LIST option ON.

C.4.5 THE ?CHECK_LIST MACRO

The **?check_list** macro is a companion to the **?list** macro. The **?check_list** macro returns the current status the **?list** macro. Thus, **?check_list** can be used to determine whether the list file is ON or OFF. The **?check_list** macro has the following syntax:

?check_list

The **?check_list** macro must abide by the following regulations:

1) **?check_list** returns the value, ON, under two conditions:

   i. if the **?list on** macro is the most recent **?list** macro called

   ii. if no **?list** macros have been called before to the **?check_list** call, and the compiler LIST option is ON.

2) **?check_list** returns the value, OFF, under two conditions:

   i. if the **?list off** macro is the most recent **list** macro called.

   ii. if no **?list** macros have been called before to the **?check_list** call, and the compiler LIST option is OFF.

The **?check_list** macro is useful when used with the **?xdef** macro to save the current state of the listing option, before turning the option off. Then the listing option can be returned to its orginal state. For example, considering the following sequence of macro calls:

```
?xdef ?prev_list_option {?check_list}
?list off
?include external_file.sol
?list ?prev_list_option
```

This sequence of calls illustrates a use for **?check_list**. The intention is to include a file, with the listing option off. Once the file is **?included**, however, the listing file is returned to its original setting. The orginal setting is saved by the first call to **?xdef**. If the listing option were ON, this call would be the equivalent of:

```
?def ?prev_list_option {ON}
```

Thus, the last line expands to: **?list ON**. Saving the state of the listing option is sometimes necessary to avoid the error of turning the listing ON when it was originally OFF.

C.4.6 THE ?COMPONENT_NAME MACRO

The ?component_name macro is useful when used with the ?xdef macro inside of ASSEMBALGEs or COMPONENTs. The ?component_name macro returns the current ASSEMBLAGE nesting in extended identifier notation. The ?component_name macro has the following syntax:

?component_name

The function of the ?component_name macro is illustrated by the following example:

```
ASSEMBLAGE outer (0, ' ')
SUMMARIZE
  sum
END SUMMARIZE

  COMPONENT one_in (1, ' ')
    COMPONENT two_in (2, ' ')
      sum = 12
    END two_in
    COMPONENT same_as_two (2, ' ')
      sum = 15
      ?xdef ?comp {?component_name}
    END same_as_two
  END one_in
  COMPONENT same_as_one (1, ' ')
    sum = sum?comp + 4
  END same_as_one
END outer
```

In this example, the ?xdef is equivalent to:

?def ?comp {@same_as_two@one_in@outer}

Thus, the call to ?comp is replaced with:

@same_as_two@one_in@outer.

In this way, sum = sum?comp + 4 becomes the SOL statement:

sum = sum@same_as_two@one_in@outer + 4

More information on the extended identifier notation, and on ASSEMBLAGEs themselves can be found in chapter 7.

C.4.7 THE ?APPEND AND ?XAPPEND MACROS

I. ?APPEND MACRO

The ?append macro appends lines of text to the end of the replacement text of an existing macro. The ?append macro is much like ?def, except that the existing replacement text is saved, and the new lines of text are tacked on .

The ?append macro has the following syntax:

?append ⟨ macro name ⟩ { ⟨ append text ⟩ }

where :

⟨ macro name ⟩	is the name of the existing macro. Consists of the symbol, ?, followed immediately by a SOL identifier.
⟨ append text ⟩	is the text to be appended on the end of ⟨ macro name ⟩ , can be any characters, with stipulations on the use of braces. This section is delimited by the open and close braces, "{" and "}."

The following rules govern how the ?append macro can be used in a SOL program:

1) An ?append macro can appear anywhere in a SOL program blanks can appear. However, an ?append CANNOT appear immediately after an ?include macro, on the same line as the ?include macro.

2) It is ILLEGAL to ?append text to an undefined macro; an error message will result.

3) It is ILLEGAL to ?append a call to a macro to itself e.g. ?append ?mac1 {?mac1} . No error message is given, and an infinite loop will result. In other words, this is BAD, don't do it. Also, don't do things like the following either:

```
?def ?a {6}
?def ?b {?a}
?append ?a {?b} !  BAD appends ?a to itself.
```

4) Macro calls in the append text are not expanded at the time of the append, just as macros in the replacement text of a ?def are not expanded until the macro being defined is called. (See section C.4 of this chapter)

5) The ?append macro appends LINES of text to the end of an existing macro; it does not append text onto the last line of an existing macro. For example,

```
?def ?a {X =}
?append ?a {12}
```

a call to ?a expands to be:

```
X =
12
```

not: `X = 12`

II. ?XAPPEND MACRO

The **?xappend** macro acts exactly like the **?append** macro, but the append text is handled somewhat differently. In an **?xappend**, all macros in the append text are expanded when the **?xappend** OCCURS. With an **?append**, macros in the append text are not expanded until the macro being appended to is CALLED. Thus, the difference between an **?xappend** and **?append** is analagous to the difference between **?xdef** and **?def**. The difference between **?xappend** and **?append** is best illustrated with an example. Consider the following:

Example 1

```
?def ?mac1 {6}
?def ?mac2 {X = 12}
?xappend ?mac2 {x = ?mac1}
?def ?mac1 {9}
?mac2
```

Example 2

```
?def ?mac1 {6}
?def ?mac2 {X = 12}
?append ?mac2 {x = ?mac1}
?def ?mac1 {9}
?mac2
```

In example 1, **?mac1** is expanded when the **?xappend** to **?mac2** occurs. Thus, the **?xappend** on the third line is equivalent to:

```
?append ?mac2 {x = 6}
```

When **?mac2** is called on the last line, it expands to be:

```
X = 12.
X = 6
```

However, in example 2, **?mac1** is expanded when **?mac2** is called, on the last line. By this time, **?mac1** has been redefined. Therefore, the call to **?mac2** in example 2 expands to be:

```
X = 12.
X = 9
```

Thus, a call to **?xappend** is equivalent to a call to **?append** except that the macros in the **?xappend**'s append text are expanded before the text is appended. Aside from the different expansion times of macros in the append text, **?xappend** and **?append** are equivalent.

C.5 SUMMARY OF MACROS

In general, all macro calls or definitions abide by the following rules:

1) A macro call is expanded nearly anywhere it appears in a SOL program, with the following exceptions:

   - Macro calls are NOT expanded when a macro is defined; the expansion occurs when the defined macro is called. For example:

```
?def ?mac1 { x = 2}
?def ?mac {?mac1}
?def ?mac1 {x = 4}
?mac
```

The call to `?mac` will expand to: `x = 4` (`?mac1` expanded). If `?mac1` had been expanded when `?mac` was defined, the replacement text would have been `x = 2`.

- Macro calls are NOT expanded inside SOL comments or SOL strings.

Note: When strings are passed as macro parameters, the quote marks are stripped off in the replacement text. (See C.2.2.2, **Example 1**, item 3) for details)

2) A user defined or predefined macro can appear anywhere in a SOL program its replacement text can legally appear. (See section C.4 of this appendix for information about predefined macros)

3) All macro expansions occur at COMPILE TIME. The result of expanding all macros will be the same as if the corresponding replacement text was typed instead of each macro call.

4) The replacement text will appear in the compiler LISTING instead of the macro call.

There are some exceptions to the above guidelines, as covered in previous sections.

Index

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-100566		

4. Title and Subtitle	5. Report Date
The Preliminary SOL Reference Manual	January 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Stephen H. Lucas and Stephen J. Scotti	
	10. Work Unit No.
	506-80-31-04

9. Performing Organization Name and Address	11. Contract or Grant No.
NASA Langley Research Center Hampton, VA 23665-5225	
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, DC 20546-0001	14. Sponsoring Agency Code

15. Supplementary Notes

Stephen H. Lucas, Vigyan Research Assoc., Inc., Hampton, Virginia.
Stephen J. Scotti, Thermal Structures Branch, Structural Mechanics Division, NASA Langley Research Center, Hampton, Virginia.

This document is a reference for work described in NASA TM 100565.

16. Abstract

The Sizing and Optimization Language, SOL, a high-level special-purpose computer language has been developed to expedite application of numerical optimization to design problems and to make the process less error-prone. This document is a reference manual for those wishing to write SOL programs. SOL is presently available for DEC VAX/VMS systems. A SOL package is available which includes the SOL compiler and runtime library routines. An overview of SOL appears in NASA TM 100565.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Optimization, nonlinear mathematical programming, computer languages, design tools, SOL	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	203	A10

NASA FORM 1626 OCT 86

For Sale by the National Information Service, Springfield, Va. 22161